Tekoäly ja go-peli

Hanne Korhonen

Tampereen yliopisto
Informaatiotieteiden yksikkö
Tietojenkäsittelyoppi
Pro gradu -tutkielma
Ohjaajat: Erkki Mäkinen ja Timo Poranen
Heinäkuu 2014

Tampereen yliopisto
Informaatiotieteiden yksikkö
Tietojenkäsittelyoppi
Hanne Korhonen: Tekoäly ja go-peli
Pro gradu -tutkielma, 54 sivua, 1 liitesivu
Heinäkuu 2014

Tässä tutkielmassa käsittelen, kuinka kiinalaista go-peliä voidaan pelata tietokoneella käyttäen erilaisia tekoälytekniikoita. Go-pelissä on suuri haarautumiskerroin eli pelitilanteissa on useimmiten mahdollista tehdä lukuisia eri siirtoja. Tämän aiheuttamat ongelmat ovat yksi syy siihen, että go-peliä pelaavat ohjelmat ovat vielä paljon huonompia kuin parhaat ihmispelaajat. Esittelen tutkielmassa muutamia tekniikoita, kuten Monte Carlo -puuhaku, joilla on päästy tämän hetken parhaisiin go-peliä pelaaviin tietokoneohjelmiin. Lisäksi käsittelen tekniikoita, joita on lisätty Monte Carlo -puuhakuun, jotta puuhakua saataisiin tehostettua toimimaan paremmin.

Avainsanat ja -sanonnat: go-peli, tekoäly, lautapeli, Monte Carlo -puuhaku.

Sisällys

1. Johdanto

Go-peli on alkuperältään kiinalainen sotastrategiapeli, joka on tullut tunnetuksi siitä, että se on yksi tekoälyn suurista haasteista. Go-peli on kaksin pelattava lautapeli, jota pystytään pelaamaan erikokoisilla neliön mallisilla laudoilla. Go-peli on suosittu ympäri maailmaa ja sitä pelataan harraste- ja kilpailumielessä. Maailman parhaat pelaajat pystyvät hankkimaan melkein miljoona Yhdysvaltain dollaria vuodessa pelaamalla go-peliä [Sensei's Library, 2014].

Tekoälylle on olemassa monenlaisia erilaisia määritelmiä, mutta Russelin ja Norwigin [2009] mukaan on olemassa neljä tavoitetta, joita tekoäly tavoittelee:

1. Systeemit, jotka ajattelevat ihmisten lailla.
2. Systeemit, jotka toimivat ihmisten lailla.
3. Systeemit, jotka pystyvät rationaaliseen ajatteluun.
4. Systeemit, jotka käyttäytyvät rationaalisesti.

Ihmisyyden tavoittelua tekoälyssä kutsutaan usein vahvaksi tekoälyksi. Se on empiirinen tiede, joka sisältää hypoteesit ja kokeelliset tulokset. Rationaalista käyttäytymistä ja ajattelua tavoittelevaa tekoälyä kutsutaan usein heikoksi tekoälyksi, koska se sisältää matematiikkaa ja konetekniikkaa, eikä edes yritä käsittää ihmisen kognitiivisia kykyjä. [Russel and Norwig, 2009] Ihmisen lailla toimivia systeemejä ei ole pystytty luomaan, ja go-pelissä ihmisen kaltaisesti toimiva systeemi olisi ihanteellinen, mutta sellaista ei olla pystytty tekemään. Go-peliä on tähän mennessä yritetty suurimmaksi osaksi pelata tietokoneilla käyttäen rationaalisia systeemejä.

Go-pelin suuri haaste tekoälylle on vähäisistä säännöistä johtuva pelin monimuotoisuus. Tavanomaiset tekoälytekniikat kuten pelipuu ja minimax-haku, joilla on pystytty voittamaan muiden lautapelien mestareita, ei toimi go-pelissä. Tutkielman tarkoitus on kirjallisuuden avulla kartoittaa, kuinka go-peliä voidaan pelata tekoälyn avulla. Tutkielmassa käydään läpi useita eri tekniikoita, mutta eniten keskitytään Monte Carlo -puuhakuun. Monte Carlo -puuhaun avulla on saatu aikaan ohjelmia, jotka pystyvät pelaamaan go-peliä hyvän harrastelijapelaajan tasolla. Tutkielmassa perehdytään myös eri tekniikoihin, joita on lisätty Monte Carlo -puuhakuun, jotta sen tehokkuutta ja pelitasoa saataisiin nostettua entisestään. Go-peliä tutkivaa ja ohjelmia luovaa tekoälyn osa-aluetta kutsutaan tietokone-goksi. Sen tutkimuksilla ollaan pystytty luomaan uusia tekoälytekniikoita, joita on pystytty käyttämään myös monien muiden alojen hyödyksi, kuten voimalaitosten hallintaan [Kroeker, 2011].

Luvussa 2 esitellään aluksi muutama yleinen tekoälytekniikka. Luvussa 3 esitellään go-peli, sen historia ja säännöt. Go-pelin monimutkaisuuteen tutustumisen jälkeen katsotaan, kuinka tietokoneet ovat menestyneet go-pelin pelaamisessa. Luvussa 4 esitellään Monte Carlo -puuhaku (MCTS), joka on pelipuun ja Monte Carlo -simulaation yhdistelmä. Luvussa esitellään myös UCT-algoritmi, joka

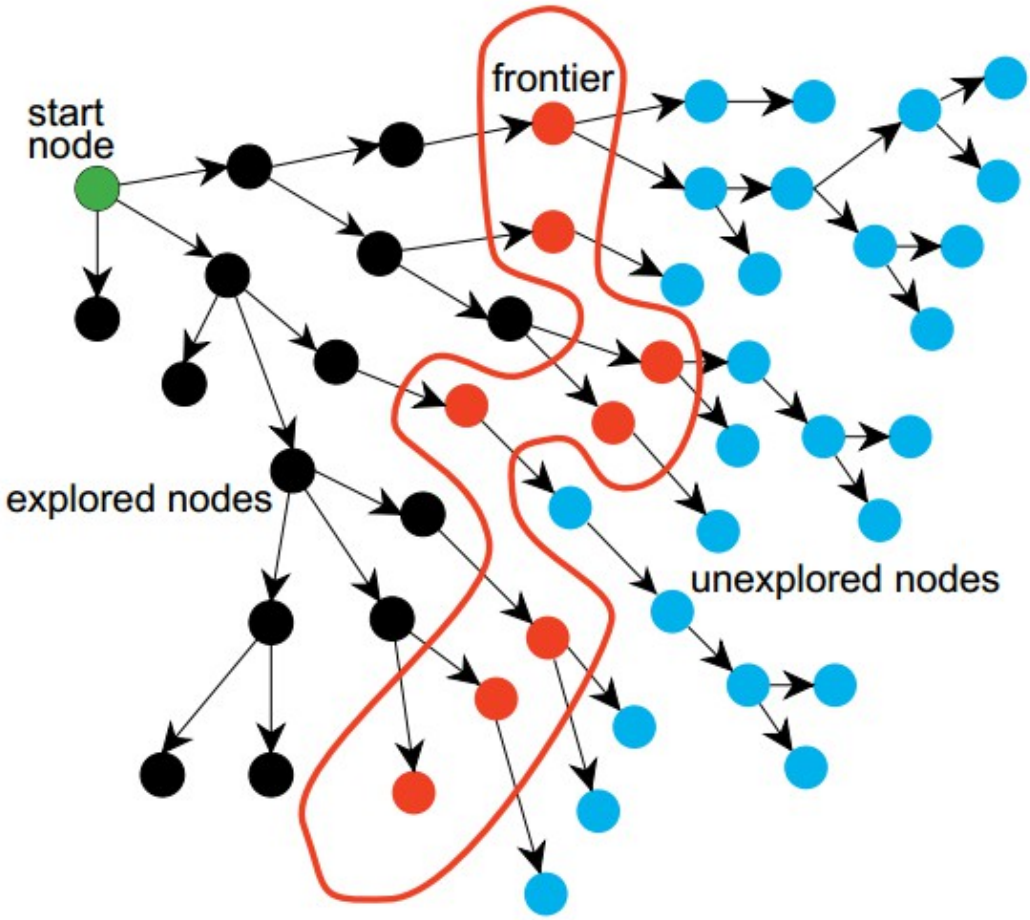
on menestyneempi versio Monte Carlo -puuhausta. Seuraavassa luvussa tutkitaan laajennuksia, joita MCTS-algoritmin kanssa on kokeiltu, jotta tietokoneiden taso go-pelin pelaamisessa paranisi. Laajennuksia ovat mm. RAVE, rinnakkaislaskenta, solmulaajennus ja meta-MCTS. Luvussa 6 käsitellään taas muita tapoja kuin MCTS-algoritmia, joilla go-peliä on yritetty ratkaista. Neuroverkkoja, agentteja ja erilaisia koneoppimisen muotoja, kuviokirjastoja sekä liikkeiden ennustamista pohditaan myös jonkin verran. Luvussa 7 pohditaan, mikä on paras tekniikka ratkaista go-peli. Yhteenvedossa tehdään johtopäätöksiä ja mietitään, miten go-peliä pelaavat ohjelmat kehittyvät.

2. Tekoälyn peruskäsitteitä

Tekoäly yrittää ymmärtää älykkäitä yksiköitä tai olemuksia [Russel and Norwig, 2009]. Toisin kuin filosofia ja psykologia, jotka myös tutkivat älykkyyttä, tekoäly haluaa ymmärtämisen lisäksi myös rakentaa näitä älykkäitä yksiköitä tai olemuksia [Russel and Norwig, 2009]. Tekoäly on osoittautunut paljon monimutkaisemmaksi alueeksi kuin alun perin luultiin, minkä myötä ideat ovat paljon odotettua rikkaampia ja mielenkiintoisempia [Russel and Norwig, 2009]. Tekoäly sisältää jo monia omia osa-aluieta, kuten ongelmien havannointi ja looginen päättelykyky. Erilaisia tekoälysovelluksia ovat muun muuassa shakin pelaaminen, matemaattisten teorioiden todistaminen, runojen kirjoittaminen sekä sairauksien diagnosointi [Russel and Norwig, 2009]. Tässä luvussa keskitytään erilaisiin perusmenetelmiiin, joita ongelmien ratkaisemiseen on käytetty. Luvun ongelmina on keskitytty varsinkin erilaisten pelien, kuten shakin, pelaamiseen, ja tekniikat ovat siis yleisiä peleihin liittyviä tekniikoita. Pelit ovat hyvä alue tutkia tietokoneiden älykkyyttä, koska peleissä on ennalta määrätyt säännöt ja tavoitteet [Doyle, 2014].

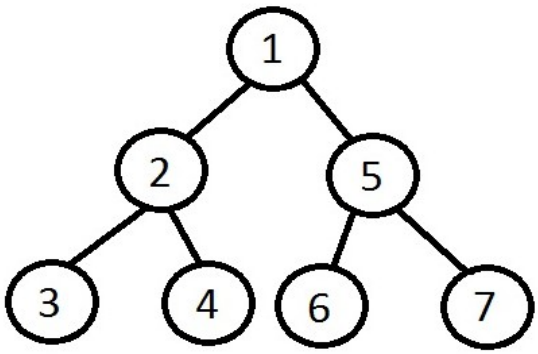
2.1. Hakualgoritmit

Hakualgoritmin tavoitteena on löytää hyvä siirto monien siirtojen joukosta. Geneerinen hakualgoritmi saa graafin, aloitussolmut sekä tavoitesolmut, joiden avulla algoritmi tutkii polkuja aloittaen aloitussolmuista [Poole et al., 2014]. Hakualgoritmi pitää yllä tietoa siitä, missä solmuissa haussa ollaan menossa. Kuvassa 1 on esitetty, kuinka haku etenee graafissa. Mustat solmut ovat jo tutkittuja solmuja, punaiset ovat niitä, joissa haku on parhaillaan menossa, ja siniset solmut ovat vielä tutkimattomia. Haku etenee graafissa riippuen siitä, mitä hakustrategiaa käytetään. Haku voi tapahtua systemaattisesti käymällä läpi kaikki solmut järjestyksessä tai solmut voidaan lajitella jonkin kriteerin mukaan.

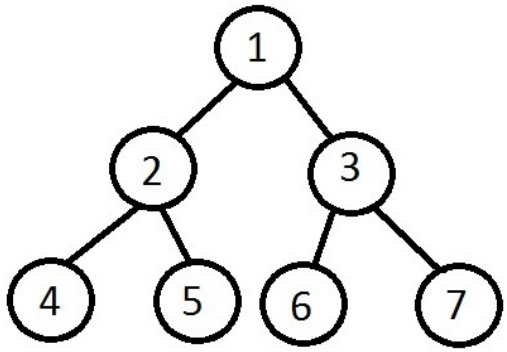


Kuva 1: Haku graafista [Poole et al., 2014].

Hakustrategioita on muun muuassa syvyys-ensin, rintamahaku ja erilaiset heuristiset haut kuten paras-ensin -hakustrategia. Syvyys-ensin -haku valitsee aina yhden haaran loppuun asti, kunnes solmut loppuvat ja se jatkaa lähimmästä tutkimattomasta haarasta tutkien lehtisolmuun asti, jolloin se taas valitsee lähimmän tutkimattoman haaran ja etenee näin koko graafin loppuun. Kuvassa 2 on esitetty hakupuussa, kuinka syvyys-ensin -haku etenee; solmut on numeroitu siihen järjestykseen, jossa syvyys-ensin -haku ne käsittelee. Syvyys-ensin -haun ongelma on se, ettei se pysähdy välttämättä koskaan loputtomissa tai syklisissä graafeissa [Poole et al., 2014]. Rintamahaku käy läpi kaikki solmut yhdeltä syvyydeltä ennen kuin se etenee uudelle syvyydelle. Solmun haarautumiskerroin on sama kuin samalla syvyydellä olevien solmujen määrä [Poole et al., 2014]. Jos haarautumiskerroin on kaikille solmuille äärellinen, niin rintamahaku löytää varmasti tavoitesolmuun [Poole et al., 2014]. Kuvassa 3 on esitetty, kuinka rintamahaku hakee solmuja hakupuusta; solmujen numeroista näkee, kuinka haku etenee ensin tutkimalla samassa syvyydessä olevat solmut ennen kuin se etenee seuraavalle tasolle.



Kuva 2: Syvyys-ensin-haku



Kuva 3: Rintamahaku

Heurististen hakujen idea on se, että otetaan huomioon tavoite, kun yritetään ratkaista ongelmaa [Doyle, 2014]. Hakua voidaan usein ohjata muulla tiedolla ja tätä tietoa kutsutaan heuristiikaksi. Paras-ensin -hakustrategia valitsee heuristiikan avulla solmun, jonka arvo on lähimpänä tavoitetta [Doyle, 2014]. Heuristiset haut eivät välttämättä toimi paremmin kuin tavalliset raakavoimahaut. Paras-ensin -hakua ei välttämättä löydä ratkaisua, vaikka sellainen on olemassa, eikä se myöskään löydä aina lyhyintä polkua ratkaisuun [Doyle, 2014].

Ongelmanratkaisu on tavoitesolmujen hakua hakuavaruudesta. Hakuavaruus on ympäristö, jossa haku tapahtuu [Doyle, 2014]. Hakuavaruus sisältää joukon ongelman tiloja ja joukon operaatioita, joiden avulla tiloja vaihdetaan [Doyle, 2014]. Hakuavaruuden voi esittää puuna. Ongelmalla on jokin alkutila, josta lähdetään liikkeelle. Kun ollaan tehty jokin operaatioista, ollaan uudessa tilassa, josta taas pystytään tekemään samat operaatiot. Jossain oikeiden operaatioiden teon jälkeen löytyy tavoitetila, jossa ongelmalle löytyy ratkaisu.

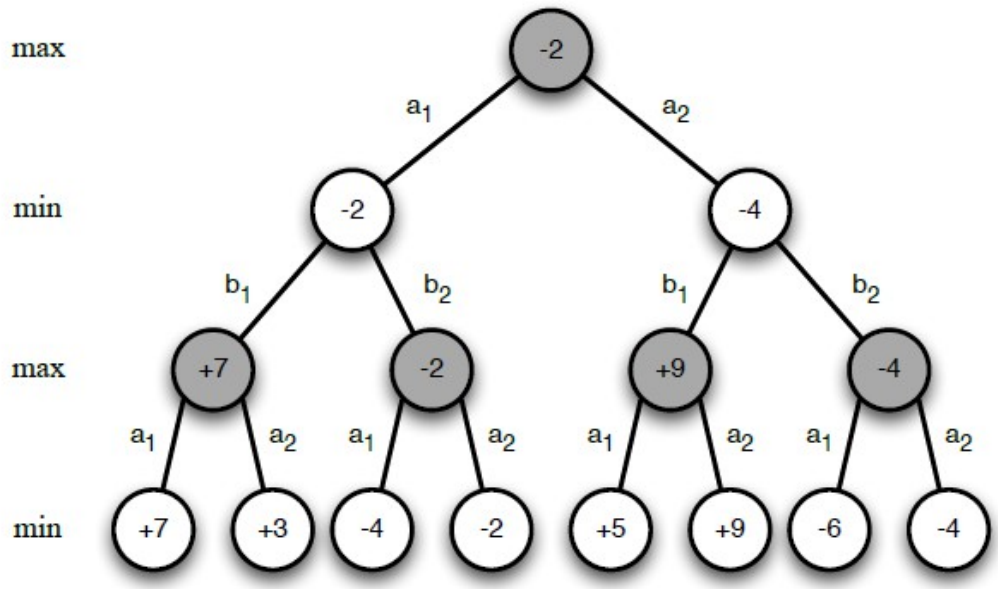
2.2. Pelipuu

Pelipuu on puumallinen tietorakenne, jota käytetään kaksinpelattaviin peleihin. Pelipuun avulla yritetään löytää mahdollisimman hyvät siirrot vastustajaa vastaan. Pelipuu järjestää mahdolliset tulevat siirrot puurakenteeksi [Gelly et al., 2012]. Molempien pelaajien siirrot on kuvattu pelipuussa, jossa joka toinen taso on vastustajan taso ja joka toinen pelaajan. Täydellinen pelipuu sisältää kaikki mahdolliset siirrot ja pelitilanteet, joilloin hyvien siirtojen hakeminen on helppoa käyttämällä jotain hakualgoritmia. Silloin kaikille lehtisolmuille on mahdollista antaa tilaksi voitto, häviö tai tasapeli.

Pelipuu sisältää kaikki puurakenteelle tyypilliset osat: juuret, kaaret ja solmut. Juuri kuvaa pelin aloitustilaa. Solmut ovat pelin tilanteita, joihin päästään aloitustilanteesta sääntöjen mukaisilla siirroilla. Samoja pelitilanteita kuvaavia solmuja voi olla useita pelipuussa, koska samoihin

pelitilanteisiin voidaan päästä useilla eri siirroilla [Gelly et al., 2012] Solmut sisältävät myös voitto- tai arviointipisteitä [Niekerk et al., 2012]. Kaaret ovat siirtoja, jotka johtavat solmujen tiloihin. Lehtisolmut ovat pelin viimeisiä mahdollisia tiloja, jolloin pelin tulos on jo selvillä. Pelipuun tasot vaihtuvat jokaisen siirron välillä. [Gelly et al., 2012; Doyle, 2014]

Pelipuuhun tehdään hakuja, jotta paras mahdollinen siirtosarja löydettäisiin [Gelly et al., 2012]. Haku voidaan tehdä muun muuassa minimax-haulla. Minimax-haku on syvyys-ensin ja syvyys-rajattu hakutekniikka. Minimax-haku etsii solmuja vain tiettyyn määrättyyn syvyyteen asti ja käsittelee näitä solmuja aivan kuin ne olisivat lehtisolmuja. Näille solmuille algoritmi määrää arvot heuristiikalla, jota kutsutaan staattiseksi evaluointifunktioksi. Sen jälkeen algoritmi kiipeää puuta takaisin juurta kohti ja määrää rekursiivisesti arvoja solmuille. Pelaajan solmuille annetaan suurin arvo solmun lapsien joukosta ja vastustajan solmuille annetaan pienin arvo solmun lapsien joukosta. Kuvassa 4 on pieni pelipuu, jolle on annettu arvot minimax-haulla. Pelajaan arvot on max-tasoilla, joissa pelaajan arvoksi yritetään saada mahdollisimman suuri arvo. Vastustajan arvot ovat min-tasoilla, joissa haetaan mahdollisimman pieni arvo vastustajalle. [Doyle, 2014]



Kuva 4: Minimax-pelipuu [Gelly et al., 2012]

Jos pelipuu on iso, minimax-haku ei ole toimiva. Alpha-beta-karsiminen on parannus minimax-hakuun ja se on suosituin hakualgoritmi pelipuissa. Alpha-beta-karsiminen mahdollistaa myös tietokoneen resurssien säästämisen sekä mahdollisuuden tehdä hakuja epätäydelliseen pelipuuhun

[Niekerk et al., 2012]. Alpha-beta-karsiminen on syvyys-ensin-algoritmi, joka kulkee puuta määritetyssä järjestyksessä, esimerkiksi vasemmalta oikealle, ja käyttää löytämäänsä tietoa oksien karsimiseen. Karsitut oksat ovat sellaisia, joiden arvo ei vaikuta juuren minimax-arvoon. Leikatut oksat sisältävät siis siirtoja, joita pelaaja aina välttää, koska parempia vaihtoehtoja on olemassa. Alpha-beta-karsimisen nimi tulee kahdesta arvosta alphasta ja betasta. Alpha-arvo on matalin raja-arvo, joka puusta on löydetty ja beta-arvo taas suurin raja-arvo. Kun leikataan oksia pelaajan tasoilta puusta, niin käytetään beta-arvoa, ja vastustajan tasoilta leikataan alpha-arvon avulla. [Doyle, 2014]

Tietokoneohjelmat, jotka ovat perustuneet eri versioihin minimax-hausta ja alpha-beta-karsimiseen, ovat voittaneet mestaripelaajat shakissa, tammessa ja othellossa [Gelly et al., 2012].

2.3. Monte Carlo -simulaatio

Monte Carlo -simulaatio on tekniikka, jota on käytetty monissa peleissä onnistuneesti, kuten othellossa, scrabblessa, ristinollassa, shakissa, backgammonissa, bridgessä sekä pokerissa. Monte Carlo -nimitys juontaa juurensa siitä, kun alun perin tekniikkaa käytettiin peleissä, joita pelattiin kasinoissa [Bouzy, 2005]. Monte Carlo -simulaatio on yksinkertainen simulaatiopohjainen -hakualgoritmi [Gelly and Silver, 2011], joka hakee tietoa tekemällä useita pelisimulaatioita.

Simulaatiopohjainen haku perustuu siihen, että se arvioi pelitilanteita tekemällä simuloituja pelejä. Simulaatiot alkavat juuresta, ja haku etenee pelitilanteita ja siirtoja läpikäyden, kunnes peli loppuu. Siirrot valitaan käyttämällä luotua simulaatiomenettelytapaa, esimerkiksi satunnaisesti, ja pelin sääntöjen avulla luodaan uusi pelitilanne. Simuloitujen pelien tuloksien avulla arvioidaan koettuja pelitilanteita ja siirtoja ja niille päivitetään uudet arvot. [Gelly and Silver, 2011]

Monte Carlo -simulaatio suunniteltiin alunperin othellon, ristinollaan ja shakkiin. 1990-luvulla sitä käytettiin backgammonin, bridgeen, pokeriin sekä scrabbleen. Monte Carlo -simulaation yleinen versio toimii niin, että tekemällä simulaatiota arvioidaan pelin pelitilannetta. Siirtoja valitaan satunnaisesti itsepeleissä, kunnes päästään pelin loppuun. Satunnaisuus ei ole oikeasti aina satunnaisuutta, vaan hyviä siirtoja valitaan jonkin heuristiikan mukaan, joista sitten valitaan satunnaisesti yksi siirto. Pelin loppuun pääseminen kuitenkin vaatii sen, että siirtojen lukumäärä on rajoitettu. Jokaisen simulaation tuloksena on vektori, joka sisältää molempien pelaajien loppuratkaisut simulaation perusteella. Pelitilanteen lopputulokset arviodaan simulaatioiden jälkeen keskimääräisiksi tuloksiksi. Monte Carlo -puuhaku ei käytä tällaista simulaatiota, vaan se kohdentaa haun hakuavaruuden lupaavimpiin paikkoihin. [Chaslot, 2010]

2.4. Monte Carlo -puuhaku

Monte Carlo -puuhaku (MCTS) on heuristinen hakualgoritmi yksin tai kaksin pelattaviin peleihin. MCTS perustuu satunnaiseen hakuavaruuden tutkimiseen [Chaslot, 2010]. Löydösten perusteella rakennetaan pelipuu [Chaslot, 2010], johon kuuluu juuri ja äärellinen määrä solmuja [Chou et al.,

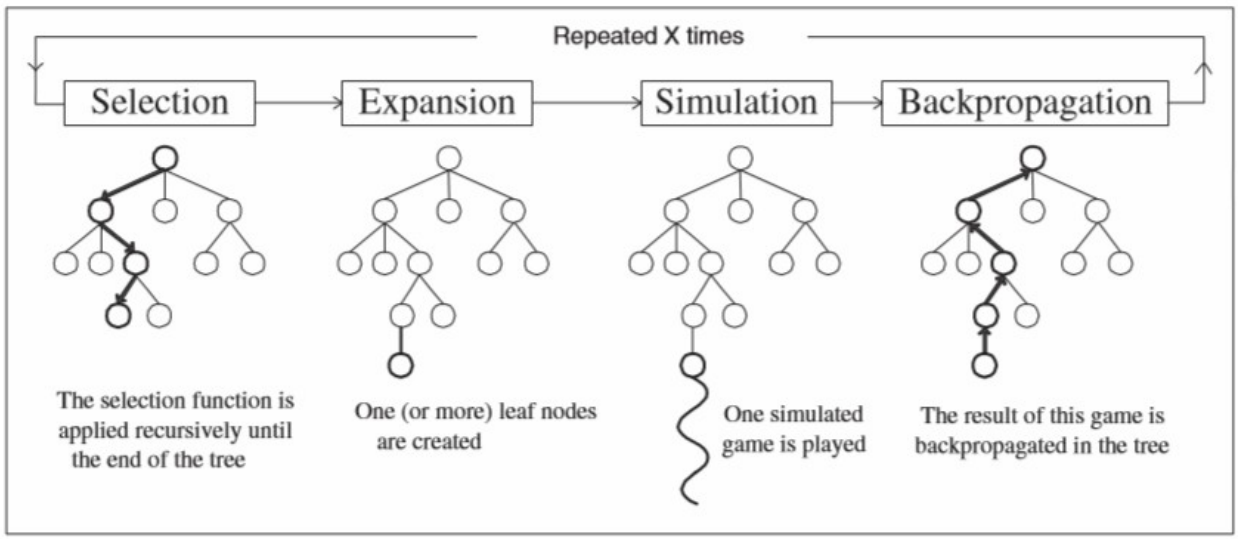
2011]. Mitä enemmän hakuavaruutta tutkitaan, sitä paremmin pystytään arvioimaan lupaavien siirtojen arvoja [Chaslot, 2010] ja näin löytämään parempia siirtoja. MCTS-algoritmista on monia hienoisesti vaihtelevia versioita. Rimmel ja muut [2010] kuvailevat MCTS-algoritmin inkrementaaliseksi puurakenteeksi, joka kuvaa mahdollisia tulevia tiloja. He jatkavat, että MCTS toimii joko maantierosvoperiaatteella (bandit formula) tai Monte Carlo -simulaatiolla. Chaslotin [2010] mukaan MCTS-algoritmi on käyttökelpoinen, jos pelin loppuratkaisut ovat rajalliset, peli on täydellisen informaation peli ja pelin pituus on rajattu, jotta simulaatiot eivät kestä liian kauan.

MCTS-algoritmi aloittaa luomalla pelipuun, joka kuvaa tämän hetkistä pelitilannetta [Niekerk et al., 2012; Chaslot, 2010]. Pelin alussa pelipuu on pelkkä juuri. Pelipuun solmut ja lehdet kuvaavat yhtä pelitilannetta, ja niiden arvot on arvioitu Monte Carlo -simulaatiolla [Gelly et al., 2012]. Jokainen solmu sisältää ainakin kaksi asiaa: nykyisen solmun pelitilanteen arvon, sekä solmussa vierailujen määrän [Chaslot, 2010]. Solmut varastoivat voittojen ja häviöiden määrät simuloiduista peleistä aloittaen solmun jälkeläisestä [Niekerk et al., 2012].

MCTS-algoritmi toimii neljässä eri vaiheessa (joita toistetaan niin kauan kuin on aikaa) [Chaslot, 2010]:

1. Valinta (Selection)
2. Laajentuminen (Expansion)
3. Simulaatio
4. Takaisin eteneminen (Backpropagation).

Valintavaiheessa käytetään valintafunktiota rekursiivisesti kunnes lehtisolmu on saavutettu. Valintafunktio voi toimia täysin satunnaisesti, mutta usein solmuja valitaan jollain tietyllä menettelytavalla. Joskus myös satunnaisuutta käytetään apuna, jos valintafunktiolla on vaikeuksia löytää sopivaa solmua. Laajentumisvaihe lisää lapsisolmun valittuun lehtisolmuun. Simulaatiovaiheessa pelataan simuloitu peli pelin loppuun asti uudesta lapsisolmusta alkaen. Takaisin etenemisessä palautetaan simulaation tulos takaisin ylöspäin puuta, kunnes juuri on saavutettu. Kuvassa 5 on kuvattu, kuinka MCTS-menetelmä toimii. Kun kaikkia neljää vaihetta on toistettu niin kauan kuin aikaa on, pelataan siirto, joka on juuren se lapisolmu, jolla on eniten vierailuja. [Chaslot, 2010;Niekerk et al., 2012; Yajima et al., 2010]



Kuva 5: MCTS:n vaiheet [Chaslot et al., 2008].

MCTS voidaan käyttää ristinollan pelaamiseen 3x3-ruudun pelilaudalla. Tälloin algoritmi aloittaa luomalla pelipuun juuren, joka kuvaa tyhjää lautaa. Tämän jälkeen aloitetaan neljän vaiheen läpikäyminen. Ensiksi luodaan lapsisolmu juurelle, joka kuvaa yhtä mahdollista siirtoa laudalle. Sen jälkeen siirron jälkeisestä tilasta luodaan simulaatio, jossa pelataan peli loppuun. Simulaation siirrot valitaan satunnaisesti hakuavaruudesta. Tämän simuloidun pelin tulos otetaan talteen ja se viedään takaisin ylöspäin puuta takaisin eteneminen -vaiheessa. Jos simuloidun pelin lopputulos on voitto, se merkitään positiivisena arvona, ja jos simuloitu peli päättyi häviöön, silloin se merkitään negatiivisena arvona; jos peli päättyi tasapeliin, merkitään tulos nollaksi. Tämän jälkeen aloitetaan vaiheiden läpikäynti uudestaan. Ristinollassa on vain kolme mahdollista ensimmäistä siirtoa (koska lauta on symmetrinen). Jolloin jos vaiheita käydään sitä useammin läpi, silloin joudutaan aina samaan solmuun uudestaan, jolloin solmuun merkitään, kuinka useasti siinä vieraillaan. Kun simuloitujen pelien tuloksia on useita, lasketaan ylemmille solmuille arvot pelitulosten keskiarvoista käyttämällä vierailumääriä. Lopullinen ensimmäinen siirto valitaan, kun vaiheita on käyty tarpeeksi läpi tai aikaa ei ole enää käytettävänä. Siirto valitaan juuren siitä lapsisolmusta, jonka arvo on paras. Tämän jälkeen luodaan uusi pelipuu, joka on pelkkä juuri. Sen jälkeen aloitetaan vaiheiden käyminen uudestaan läpi. Tämä toistuu niin kauan kuin päästään pelin loppuun.

2.5. Agentit ja koneoppiminen

Agentteja on olemassa erilaisia ja niitä voi käyttää eri tavoin. Pääasia kuitenkin on se, että agenttien tarkoitus on auttaa käyttäjää valitsemaan tämän haluamat toiminnot ja opastaa käyttäjää käyttämään ohjelmaa. On myös sellaisia agentteja, jotka tekevät valinnat käyttäjän puolesta kokonaan ilman käyttäjän syötteitä. Agentiksi voi myös kutsua oppivaa yksikköä ohjelmasta. Agentti on jotain

sellaista, joka havaitsee ja toimii [Russel and Norwig, 2009]. Agentti toimii jossain tietyssä ympäristössä. Älykäs agentti käyttäytyy älykkäästi. Agentin tapa toimia on sopiva sen tavoitteelle ja ympäristön olosuhteille. Agentti on myös joustava, jos sen tavoite tai ympäristö muuttuu kesken tehtävän. Agentti oppii kokemuksesta. Agentilla on siis aikaisempi tietämys, vanhat kokemukset, tavoitteet sekä omat huomioinnit. Näiden asioiden avulla agentti tekee päätöksen siitä, kuinka toimia seuraavaksi. [Poole et al., 2014]

Oppiminen on kyky parantaa omaa toimintaa kokemuksien avulla. Vahvistava oppiminen (reinforcement learning) on yksi tapa oppia lisää. Siinä agentti päättelee palkintojen ja rangaistusten avulla, mikä on paras tapa toimia [Poole et al., 2014]. Silverin ja muiden [2012] mukaan vahvistavaa oppimista ajatellaan usein hitaaksi prosessiksi, koska oppiminen tapahtuu usein monien kuukausien tuloksena, mutta heidän mielestään agentin on mahdollista oppia ja suunnitella suurissa ja vaativissakin ympäristöissä (kuten go-pelissä) nopeasti. Vahvistava oppimiseen sisältyy kaksi perusongelmaa: oppiminen ja suunnittelu. Oppimisvaiheessa agentti kehittää menettelytapaa olemalla vuorovaikutuksessa ympäristön kanssa. Agentti oppii tekemällä toimenpiteitä ja tutkimalla niiden seurauksia. Suunnitteluvaiheessa agentti yrittää kehittää menettelytapaa niin, ettei se enää ole vuorovaikutuksessa ympäristön kanssa, käyttäen harkintaa. Suunnitteluvaiheessa ympäristönä toimii malli ympäristöstä, ja agentti simuloi toimenpiteitä mallissa ja tutkii niiden seurauksia. Oppimisessa ja suunnittelussa voidaan käyttää samoja algoritmeja. [Silver et al., 2012]

Väliaikainen ero -oppiminen (temporal-difference learning) on yksi parhaita ratkaisuja vahvistavan oppimisen ongelmaan. Sen avulla on pystytty luomaan ohjelmia, jotka voittavat maailman parhaat pelaajat shakissa, tammessa sekä backgammonissa. Tämän on mahdollistanut arvofunktio (value function), jota harjoitetaan offline-peleissä, kun ohjelma pelaa itsekseen. Arvofunktion kehittäminen itsepeleissä saattaa kestää viikkoja tai jopa kuukausia. Väliaikainen ero -oppiminen on metodi, jolla arvioidaan menettelytapaa. Arvofunktiota päivitetään käyttämällä arvioita tulevista arvioista yleistämään tilojen välejä. [Silver et al., 2012]

3. Go-peli

Go-peli on muinainen peli ainakin 3000 vuoden takaa. Peli on lähtöisin Kiinasta, jossa sitä kutsutaan nimellä Wei-qi. Go-sana tulee japanin kielisestä sanasta Igo, joka tarkoittaa ympäröivää lautapeliä. Go-peli on toiseksi pelatuin peli maailmassa, ensimmäisenä on kiinalainen shakki. [Sensei's Library, 2014]

Go-peli pelataan laudalla, joka koostuu horisontaalisista ja vertikaalisista viivoista, jotka muodostavat neliönmuotoisen ruudukon. Aloittelijat voivat aloittaa laudoilla, joiden koko on 9x9 tai 13x13. Standardikoko on kuitenkin 19x19-lauta. Viivojen yhteyskohtia kutsutaan pisteiksi (point), joihin pelattavat kivet asetetaan. Viereisiä pisteitä ovat sellaiset pisteet, jotka on vierekkäin ja niitä yhdistää viiva. Pelin pääidea on valloittaa mahdollisimman suuri alue pelilaudalta ympäröimällä se omilla kivillään. [Sensei's Library, 2014]

Go-peli on täydellisen informaation peli, eli molemmat pelaajat tietävät kaiken, mitä pelissä voi tietää. Sattumalla tai onnella ei ole siis minkäänlaista tekemistä go-pelin tuloksissa. Kaikki on kiinni siitä, kuinka hyvin pelaaja osaa pelata peliä. Go-pelaajat jaotellaan eri taitotasoihin heidän kykyjensä mukaan. Aloittelevat pelaajat aloittavat tasolta 30 kyu ja pyrkivät kohti 1 kyu:ta [Harré et al., 2011]. Sen jälkeen alkaa harrastelijoiden dan-tasot, joista alin 1. harrastelijadan ja ylin 7. harrastelijadan. Useimmat innostuneet go-pelin pelaajat pystyvät pääsemään 1 harrastelijadanin tasolle viimeisintään muutamassa vuodessa [Harré et al., 2011]. Harrastelijatasojen jälkeen tulevat ammattilaistasot, jotka ovat myös nimeltään dan. Ammattilaisdanit jaetaan myös niin, että 1 on alin ja 9 on huippuammattilainen. Huippuammattilaisen tasolle pääseminen saattaa kestää vuosikymmeniä, eivätkä kaikki pelaajat pysty koskaan hankkimaan sitä tasoa [Harré et al., 2011]. Oman tason saa selville pelaamalla muita pelaajia vastaan, jos voittaa tietyn tasoisen pelaajan ilman tasoitusta, voi olettaa, että itsekin on tällä tasolla.

Tässä luvussa esitellään ensin lyhyt katsaus go-pelin historiaan, jonka jälkeen kerrataan go-pelin säännöt. Seuraavaksi katsotaan shakin ja go-pelin yhtäläisyyksiä ja eroja sekä go-pelin monimutkaisuutta. Lopuksi esitellään tietokone-go, joka tarkoittaa tietokoneohjelmia, jotka pelaavat go-peliä.

3.1. Historia

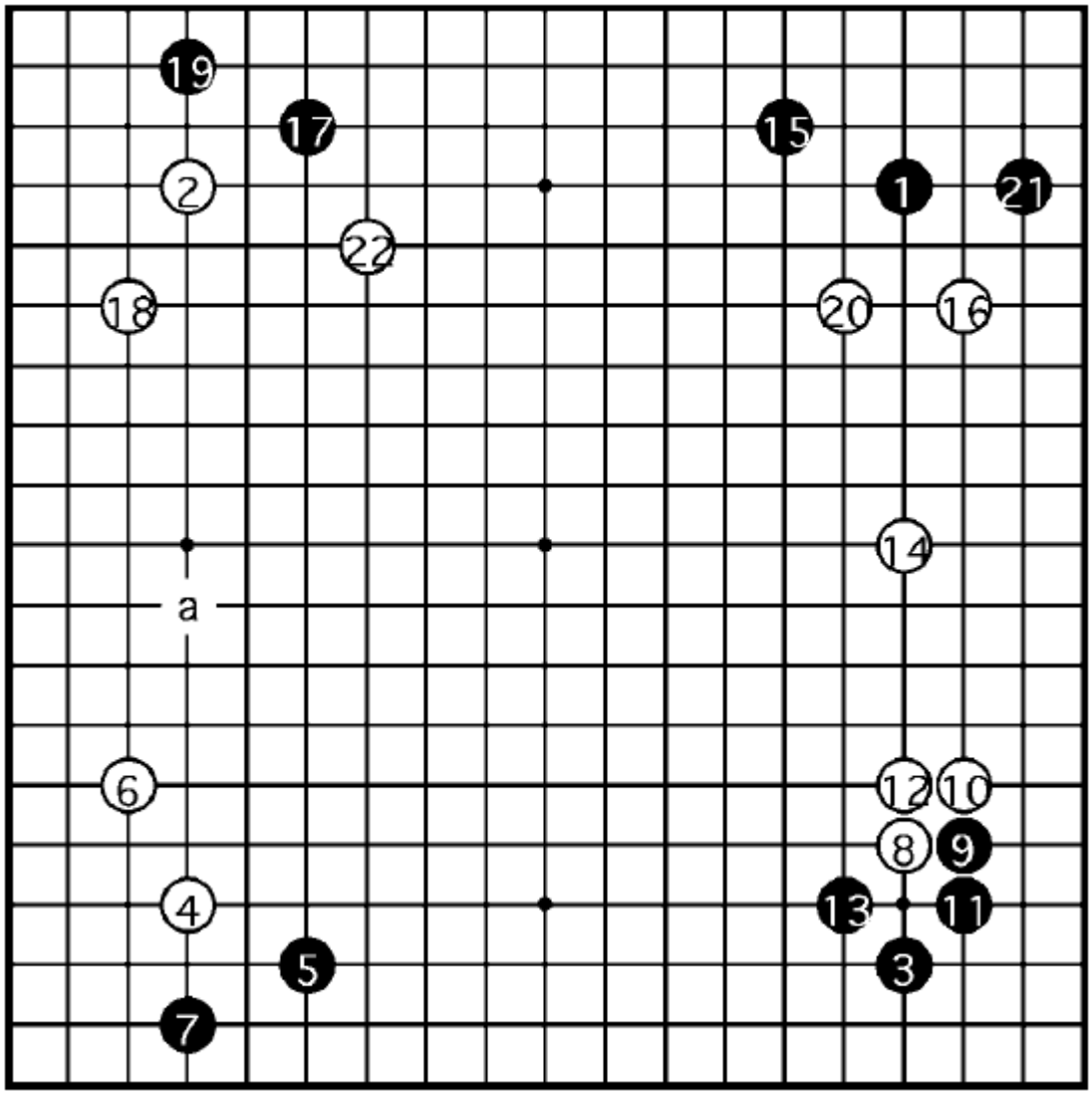
Kiinalaisen legendan mukaan Kiinan keisari Yao (2357-2255 eaa.) käski neuvonantajaansa Shunin luomaan pelin, joka opettaisi keisarin pojalle taktiikkaa ja strategiaa. Shun loi go-pelin, jonka keisarin poika hylkäsi nopeasti sanoen, että aloittava pelaaja voittaa aina. Tästä keisari suuttui ja poisti oman poikansa perimysjärjestyksestä asettaen neuvonantaja Shunin tämän tilalle. Tämä legenda on todennäköisesti Han-ajan go-pelaajien keksimä tarina. [Sensei's library, 2014]

Vanhin löydetty go-pelistä kertova maininta on vuosisadalta 600 eaa. [Sensei's Library, 2014]. Go-peli on siis keksitty ennen 600 eaa., mutta varmaa ajankohtaa ei ole pystytty päättelemään. Vuodelta 100 jaa. on löydetty ensimmäinen teksti, joka on kirjoitettu go-pelistä, “Go-pelin olemus”, jonka on kirjoittanut Ban Gu [Sensei's Library, 2014]. Ensimmäinen tallennettu go-peli on noin vuodelta 250, jossa mustilla kivillä pelasi Sun Ce ja valkoisilla kivillä pelasi Lu Fan [Sensei's Library, 2014].

Aasian ulkopuolelle go-peli levisi vuonna 1615. Italialainen jesuiitta Matteo Ricci kirjoitti päiväkirjoja, joissa hän selittää Wei-qi-pelin säännöt. Ricci vietti 30 vuotta lähetyssaarnajana Kiinassa. Viisi vuotta hänen kuolemansa jälkeen päiväkirjat käännettiin latinaksi ja julkaistiin kirjana. Kirjasta tuli todella suosittu ja se käännettiin monille Euroopan kielille muutaman vuosikymmenen aikana. Kuitenkaan eurooppalaiset eivät osanneet pelata go-peliä, eikä heillä ollut tiedossa kaikkia sääntöjä ennen kuin vasta 1900-luvulla. Ensimmäinen länsimaalainen ammattilaispelaaja oli Manfred Wimmer (1944-1995), josta tuli 1 danin ammattilainen vuonna 1978. [Sensei's Library, 2014]

3.2. Säännöt

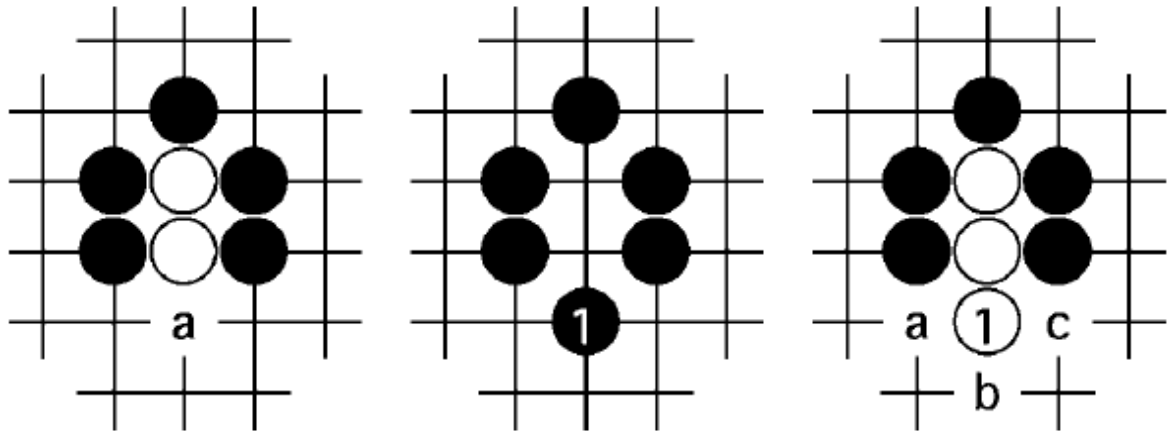
Go-pelin säännöt ovat yksinkertaiset. Pelissä on kaksi pelaajaa, jotka pelaavat vuoron perään ja asettavat laudalle nappuloita, joita kutsutaan kiviksi. Molemmilla pelaajilla on oman väriset kivet, joko mustat tai valkoiset. Tässä tutkielmassa mustilla kivillä pelaavaa pelaajaa kutsutaan nimellä musta ja valkoisilla kivillä pelaajaa on valkoinen. Musta aloittaa pelin, ja usein musta pelaaja on tasoltaan huonompi pelaaja. Go-peli alkaa tyhjältä laudalta, jolloin pelaaja voi aloittaa pelin mihin tahansa 361 paikasta laudalta. Kuvassa 6 on tyypillinen aloitus go-pelissä. [Sensei’s Library, 2014]



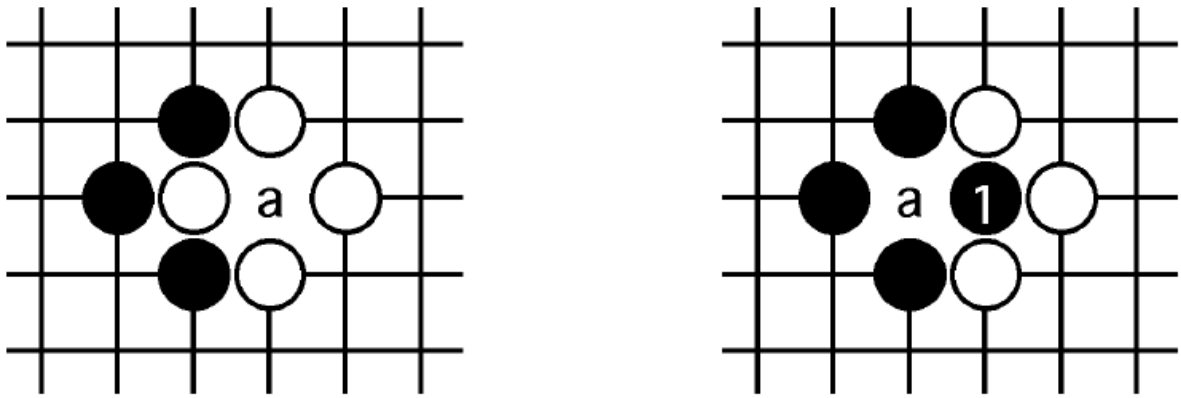
Kuva 6: Go-pelin tyypillinen aloitusvaihe [Müller, 2000]

Kiven asettamista laudalle sanotaan siirroksi. Kumpikin pelaaja voi milloin tahansa olla pelaamatta vuoroaan ja passata vuoron takaisin toiselle. Kun kiven asettaa laudalle, niin ei sitä enää saa siirtää laudalla. Kivi voi poistua laudalta vain silloin, kun se vangitaan. Go-pelissä pystytään vangitsemaan vastustajan kiviä niin, että ympäröidään vastustajan kivi omilla kivillä kokonaan. Kuvassa 7 on esimerkki vangitsemisesta. Tällöin vangittu (valloitettu) kivi poistetaan laudalta. Kivi on silloin valloitettu, jos sen ympärillä ei ole yhtään tyhjää pistettä tai samanväristä kiveä. Go-pelissä toisto on kielletty. Edellisen vuoron lautaa ei saa toistaa omalla kiven asettamisellaan. Tätä kutsutaan ko-säännöksi. Kuvassa 8 on esimerkki ko-säännön toiminnasta. Valkoinen pelaaja ei saa

tällä vuorolla asettaa kiveään kohtaan a, vaan hänen pitää pelata jonnekin muualle ensiksi.[Sensei's Library, 2014]



Kuva 7: Esimerkki vangitsemisesta [Müller, 2000]



Kuva 8: Ko-sääntö [Müller, 2000]

Peli loppuu silloin, kun molemmat pelaajat passaavat peräkkäin, tai jos toinen pelaaja luovuttaa pelin. Se pelaaja, joka hallitsee yli puolta laudasta, voittaa pelin. Loppupisteitä voidaan laskea usealla eri tavalla, mutta yleensä niistä tulee sama lopputulos. Kiinalainen tapa laskea pisteitä on yleisin [Lubberts and Miikkulainen, 2001]. Pisteet lasketaan laskemalla yhteen valloitetut vastustajan kivet ja kaikki tyhjät alueet laudalta, jotka on kokonaan ympäröity omilla kivillä [Lubberts and Miikkulainen, 2001]. Japanilainen pisteiden lasku tapahtuu laskemalla yhteen valloitetut kivet, ympäröidyt tyhjät alueet sekä lisäämällä lukuun vielä pelaajan omat kivet, jotka laudalla on [Lubberts and Miikkulainen, 2001]. Pelin lopussa valkoinen pelaaja saa yleensä

ylimääräisiä pisteitä, koska hän pelaa toisena ja on näin huonommassa asemassa. Näitä ylimääräisiä pisteitä kutsutaan komiksi. [Sensei's Library, 2014]

Aloitusvaihetta kutsutaan nimellä fuseki [Richards et al., 1998]. Siinä pelaajat tekevät alustavat alueen valtaukset laudalta. Peli alkaa usein laudan nurkista, jossa tiettyjä siirtosarjoja kutsutaan nimellä joseki [Richards et al., 1998]. Aloitusvaihe kestää 30-60 siirtoa, jonka jälkeen lauta on jakautunut useisiin alueisiin, joista pelaajat yrittävät vallata alueita. Huonomman aloituksen tehnyt pelaaja joutuu ottamaan riskejä, jotta hän voi voittaa pelin. Riskeihin kuuluu mm. hyökkääminen ja valtaaminen. Aloituksen merkitys on siis suuri. [Sensei's Library, 2014]

Go-pelin yksi hienous on se, ettei pelaajien välinen tasoero vaikuta peliin. Pelistä saa tasaväkisen niin, että huonommalle pelaajalle annetaan tasoituskiviä. Pelin alussa mustalla pelaajalla on asetettu sopiva määrä tasoituskiviä pelilaudalle strategisiin paikkoihin. Silloin myös komin määrä voidaan pienentää, jotta saadaan tarpeeksi tasoitusta. [Sensei's Library, 2014]

3.3. Shakki ja go

Go-peliä usein verrataan shakkiin ja siksi esittelen shakin ja go-pelin yhtäläisyydet ja erot. Pelit eivät ole kovin samanlaisia pelillisesti. Tekoälyn kannalta go-peli on vuosikymmeniä shakkia jäljessä, koska jo vuonna 1997 pystyttiin voittamaan shakin maailmanmestari tietokoneohjelmalla.

Suurin yhtäläisyys peleillä on se, että ne usein luokitellaan samaan kategoriaan sotastrategiapeleinä. Pelillisesti shakki ja go-peli eroavat paljon toisistaan. Molemmat ovat täydellisen tiedon pelejä, joissa molemmilla pelaajilla on kaikki informaatio tämän hetkisestä pelitilanteesta. Shakki ja go ovat molemmat strategisia pelejä, ja pelin lopputulokseen ei satunnaisuudella tai sattumalla ole mitään tekemistä. [Sensei's Library, 2014]

Eroja löytyy varsinkin pelien säännöistä. Go-pelissä kaikki kivet (eli pelinappulat) ovat samanlaisia toiminnaltaan ja niitä ei laudalle asettamisen jälkeen siirretä. Shakissa pelinappulat ovat jo valmiiksi laudalla ja nappuloilla on erilaisia sääntöjä ja liikkumistapoja. Shakissa on myös yksi tavoite, jonka avulla peli voitetaan, kun taas go-pelissä voitto saavutetaan hankkimalla pisteitä [Sensei's Library, 2014]. Go-pelissä laudalla olevat kivet lisääntyvät pelin edetessä, kun taas shakissa usein pelinappuloiden määrä vähenee.

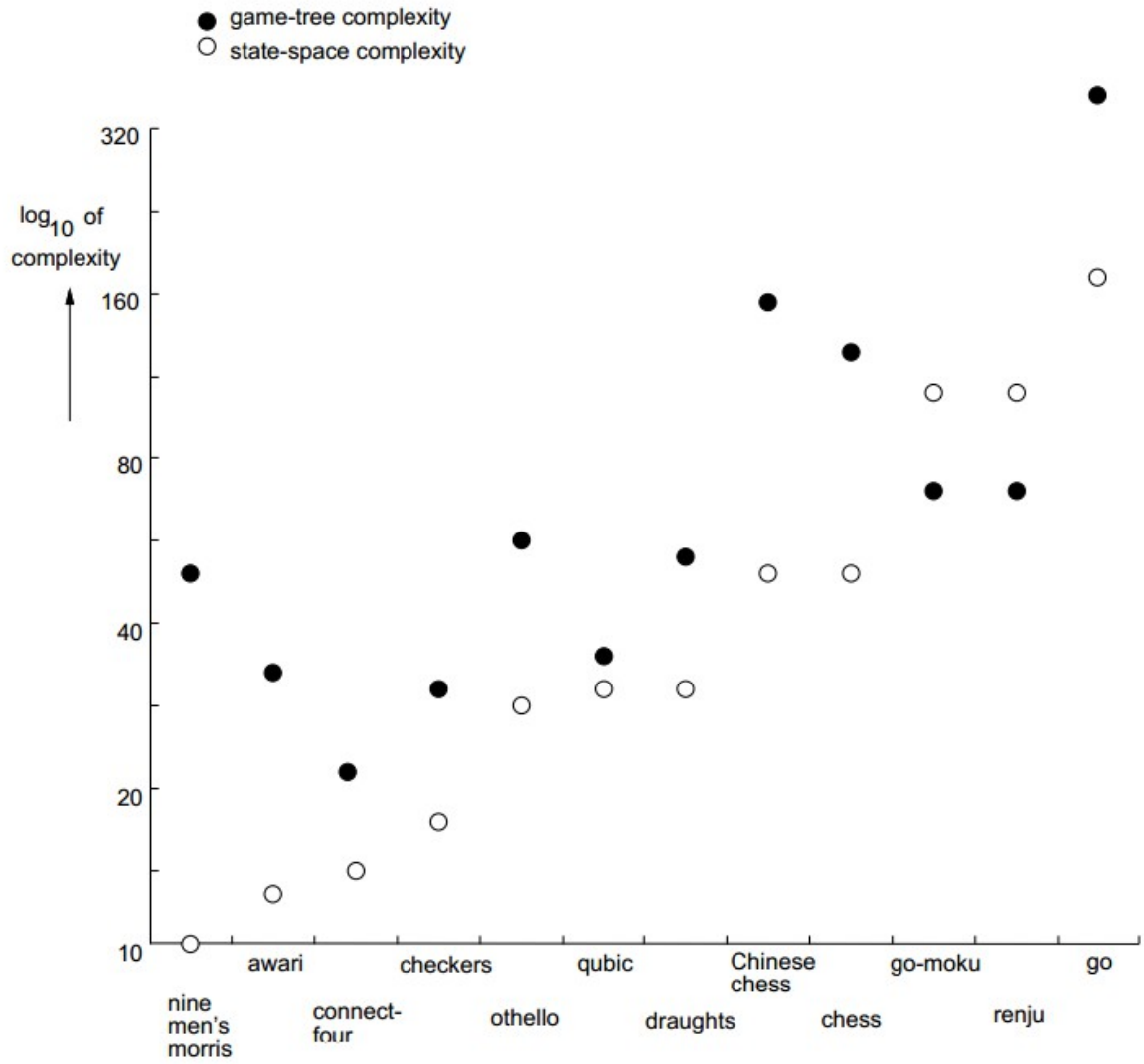
3.4. Go-pelin monimutkaisuus

Go-peli on yksinkertaisten sääntöjen lisäksi tai takia kuitenkin hyvin monimutkainen. Harré ja muut [2011] tutkivat, kuinka harrastajapelaajista tulee ammattilaispelaajia. He tutkivat 160 000 pelaajan tietoja peleistä, joissa samantasoiset pelaajat pelasivat vastakkain. Harré ja muut [2011] valitsivat 7x7-alueen laudan nurkasta ja tutkivat siihen tehtyä siirtoja. He huomasivat, että jokaisella tasolla oli noin 15 erilaista aloitussiirtoa tämän 7x7-alueen sisällä (siirrot, jotka eivät tapahtuneett alueella, jätettiin huomioimatta). Tämä jo kertoo siitä, kuinka paljon valinnanvaraa ja erilaisia vaihtoehtoja go-pelissä on. Mahdollisia pelipositioita 19x19-laudalla on $3^{19x19} \approx 10^{170}$, joista alkutilanteesta

voidaan saavuttaa n. 1,2 % [Müller, 2002]. Järjestys, jossa kivet on laitettu pelilaudalle, on epärelevanttia tietoa pelitilanteen arvioimisen kannalta [Harré et al,. 2011]. Järjestyksellä saattaa kuitenkin olla strategista merkitystä, kun suunnitellaan seuraavia siirtoja [Harré et al,. 2011].

Pelipuun monimutkaisuus on luku, joka ilmoittaa pelaajan mahdollisten siirtojen lukumäärän pelin aikana. [Sensei's Library, 2014]. Tila-avaruuden monimutkaisuus on luku, joka kertoo laillisten siirtojen määrän pelin alkutilanteesta lähtien [Sensei's Library, 2014]. Go-pelin pelipuun monimutkaisuus on 10^{360} , kun keskimääräinen haarautumiskerroin on 250 ja keskimääräinen pelin pituus on 150 vuoroa [Allis, 1994]. Tila-avaruuden monimutkaisuus (state-space complexity) on $3^{361} \approx 10^{172}$ [Allis, 1994]. Kuvassa 9 näkyy, kuinka paljon monimutkaisempi go-peli on verrattuna muihin lautapeleihin. Pelipuun monimutkaisuus ja tila-avaruuden monimutkaisuus ovat molemmat suurempia kuin missään muussa kuvassa esitellyssä pelissä.

Tromp ja Färneback [2006] laskivat tarkkaa lukua go-pelin tila-avaruuden monimutkaisuudelle. Laillisia siirtoja 19x19-laudalle ei ole vielä mahdollista laskea tarkasti nykyisillä tietokoneilla. He onnistuivat pääsemään 17x17-laudan siirtoihin asti ja tämäkin vei noin 8000 CPU tuntia ja 400 GB muistitilaa. Siirtojen lukumäärä oli 137 numeron pituinen luku. Tromp ja Färneback [2006] arvioivat, että 19x19-laudan laillisten siirtojen määrä on 171 numeroa pitkä luku.



Kuva 9: Eri pelien pelipuu ja tila-avaruuksien monimutkaisuuksien vertailua [Allis, 1994]

3.5. Tietokone-go

Go-pelin pelaaminen on tietokoneille niin vaikeaa, etteivät parhaat tietokone-go-ohjelmat pysty pelaamaan maailman parhaita go-pelaajia vastaan samalla tasolla. Go-pelin säännöt ovat kuitenkin tarpeeksi yksinkertaiset niin, että neljävuotias lapsi pystyy ne oppimaan ja pelaamaan [Sensei's Library, 2014]. Pelin ratkaiseminen tarkoittaa sitä, että kaikki mahdolliset pelitilanteet on pystytty asettamaan vaikkapa pelipuuhun, ja näin parhaan mahdollisen siirron pystyy aina hakemaan puusta. Ratkaistussa pelissä lopputulos on selvillä alusta alkaen, kun molemmat pelaajat pelaavat niin hyvin kuin mahdollista. Go-peliä ei ole ratkottu kuin 6x6-lautaan asti [Chou et al., 2011]. Go-pelin ratkaisuun on vielä pitkä matka, koska 6x6-lauta ei ole edes yleinen pelattava lautakoko go-pelissä.

Go-peliä on yritetty ratkoa jo vuosikymmeniä. Müllerin [2000] mukaan go-peli on heti shakin jälkeen eniten tutkittu ja ohjelmoitu peli. Se on vaikeaa tietokoneille, koska pelipuun haarautumiskerroin (branching factor) on suuri, peli perustuu kuviotietämykseen ja sisältää monia keskenään vuorovaikutuksessa olevia tavoitteita. Vaikeuden peliin tuo myös se, että joidenkin siirtojen seuraukset saatetaan nähdä vasta satojen siirtojen jälkeen [Silver et al., 2012]. Tietokoneiden on hankala siksi arvioida siirtojen tulevia hyötyjä, jotka ihmispelaajat pystyvät arvioimaan joskus tiedostamattaankin. Go-pelissä on kolme vaihetta: pelin aloitus, keskiosa ja loppu. Go-ohjelmissa on usein joseki-tietokantoja, joissa on tallennettuna 5000-50000 siirtoa. Hyvien josekien valitseminen on tietokoneille hankalaa, mutta vaikka tietämys hyvästä aloituksesta puuttuu, tekevät monet go-peliä pelaavat ohjelmat hyviä aloituksia tietokantojen avulla. Keskiosa on go-pelin lopputuloksen kannalta tärkein osa. Se on myös vaikein tietokoneille. Loppu on helpoin osio, mutta lopun hyvällä pelillä ei usein ole enää suurta merkitystä lopputulokseen. [Richards et al., 1998]

Ensimmäiset ohjelmat tiesivät vain go-pelin säännöt ja päättelivät kuviotietämyksen avulla, kuinka ympäröidään ja vallataan alueita [Müller, 2000]. Ensimmäisen tietokone-go-ohjelma, joka pelasi kokonaisia pelejä, ohjelmoi Albert L. Zobrist vuonna 1968. [Sensei's Library, 2014]. Ohjelma perustui go-pelin kuvioiden tunnistukseen ja jakoi pelin neljään eri vaiheeseen, joilla kaikilla oli omia kuvioita, joiden avulla oikea siirto löydettäisiin [Sensei's Library, 2014].

Tietokone-go-ohjelmat lisääntyivät huomattavasti 1980-luvun puolivälissä, kun tietokoneet muuttuivat edullisiksi ja International Computer Go Congress -yritys ryhtyi rahoittamaan tietokone-go-turnauksia [Müller, 2000]. Sen sukupolven ohjelmat, kuten Goliath, Go intellect ja Handtalk, voittivat vastustajansa paremmalla hyökkäämis- ja puolustustietämyksellä [Müller, 2000]. Tämän ajan peleissä parempi ohjelma usein voitti useilla pisteillä [Müller, 2000]. Seuraavassa vaiheessa ohjelmat, jotka ovat hyviä valtaamaan alueita, mutteivät kovin hyviä pelaajien välisissä alueiden taisteluissa, pystyivät voittamaan suuren osan peleistä. Go4++ on ensimmäinen ohjelma, joka on hyvä valtaamaan alueita, jopa uhraamalla pienen joukon aluetta saavuttaakseen voiton. Vuoteen 2000 mennessä nämä molemmat tavat olivat yhdistyneet niin, että niitä oli vaikeaa erottaa toisistaan. [Müller, 2000]

1990-luvun ohjelmia olivat Goliath, Go Intellect, Handtalk, Goemate, Go4++, Many Faces of Go, KCC IGO, Haruka, Wulu, FunGo, Star of Poland ja Jimmy [Müller, 2000]. Myös GNU GO -ohjelmasta, joka on ensimmäinen avoimen lähdekoodin ohjelma, julkaistiin ensimmäinen vakaa versio jo vuonna 1989 [Sensei's Library, 2014].

Vuonna 1993 tehtiin ensimmäinen Monte Carlo -simulaatiota hyväksikäyttävä ohjelma Gobble. B. Bruegmannin tekemä Gobble ei ollut paras ohjelma siihen aikaan, mutta se osasi pelata go-peliä ilman minkäänlaista go-peli tietämystä. Vuonna 2006 Kocsis ja Szepesvari kehittävät UCT-algoritmin. Samana vuonna CrazyStone-ohjelma voitti tietokoneolympialaisten 9x9-laudan go-

tapahtuman käyttämällä UCT-algoritmia. Vuoden 2006 jälkeen on keskitetty Monte Carlo -puuhakumenetelmiin (MCTS-menetelmiin), muun muuassa Zen, MoGo, Fuego ja CrazyStone toimivat nykyisin jonkinlaisella MCTS-metodilla. Mobiililaitteilla toimivat tietokone-go-ohjelmat eivät pärjää tällä hetkellä kuitenkaan edes vahvoille aloittelijoille, koska vähäisen muistin ja tehon takia MCTS-metodin toiminnan tehokkuus huononee paljon. [Sensei's Library, 2014]

Tietokone-go-ohjelmien tasoa on vertailtu usein laittamalla ne pelaamaan keskenään tai sitten pelaamalla ihmispelaajia vastaan. Ammattilaistason tietokone-go-ohjelmia ei ole vielä olemassa. Taulukossa 1 on koottu tietokoneiden voittoja ammattilaispelaajia vastaan 9x9-laudalla. Vuonna 1997 Janice Kim, yhden danin ammattilainen, hävisi Many Faces of Go -ohjelmalle, kun ohjelmalla oli tasoituksena 3 kiveä [Computer Go, 2014]. On kuitenkin kiisteltävissä, voiko voittoa edes laskea, koska Janice Kim pelasi samaan aikaan muitakin pelejä, eikä keskittynyt täysillä tähän peliin [Computer Go, 2014]. Seuraava voitto 9x9-laudalla tapahtui vasta vuonna 2007, jolloin Guo Juan, 5 danin ammattilainen, hävisi ilman tasoitusta MoGo-ohjelmalle, kun molemmilla pelaajilla oli kymmenen minuuttia aikaa pelata omat vuoronsa. [Gelly et al., 2012; Computer Go, 2014]. Juan pelasi kuitenkin kolme peliä MoGo-ohjelmaa vastaan, mutta hävisi vain yhden erän [Computer Go, 2014].

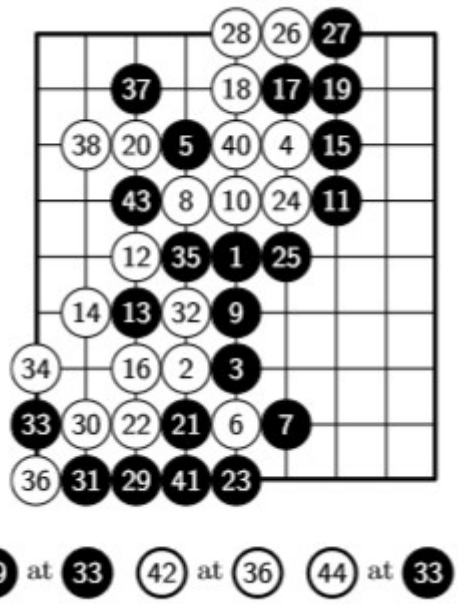
Vuosi	Ohjelma	Vastustaja	Vastustajan taso	Tasoitus
1997	Many Faces of Go	Janice Kim	1 dan	3 kiveä
2007	MoGo	Guo Juan	5 dan	Ei tasoitusta
2008	MoGo	Catalin Taranu	5 dan	Ei tasoitusta
2008	Crazy Stone	O Meien (Wan Mingwan)	9 dan	Ei tasoitusta
2008	MoGo	Catalin Taranu	5 dan	Ei tasoitusta
2009	MoGo v.4.86	Javier-Aleksi Savolainen	5 dan	Ei tasoitusta
2009	MoGoTW	Chun-Hsun Chou	9 dan	Ei tasoitusta
2009	Fuego	Zhou Junxun	9 dan	Ei tasoitusta

Taulukko 1: 9x9-laudalla pelattuja voittoja tietokoneohjelmille

Vuonna 2008 Catalin Taranu, 5 danin ammattilainen, pelasi MoGoa vastaan ilman tasoitusta. [Gelly et al., 2012; Computer Go, 2014]. Myös Taranu pelasi kolme peliä MoGoa vastaan, joista MoGo voitti vain yhden pelin [Computer Go, 2014]. Samana vuonna O Meien (Wan Mingwam), 9 danin ammattilainen, hävisi Crazy Stone -ohjelmaa vastaan ilman, että ohjelma sai tasoitusta [Computer Go, 2014]. Tämän voiton kuitenkin moni tutkija tuntuu sivuuttavan jostain syystä. Gelly

ja muut [2012] eivät listanneet sitä, vaikka Meien on 9 danin huippuammattilainen, kuten myös Zhou Junxun, joka hävisi Fuegolle vuonna 2009. He jopa ylistävät Fuegon olevan ensimmäinen ohjelma, joka on voittanut 9 danin ammattilaisen, vaikka Fuego on ensimmäinen ohjelma, joka on pystynyt tekoon virallisessa pelissä [Sensei's Library, 2014]. Kuvassa 10 on Fuego vastaan Junxun pelin lopputilanne, jossa Fuegoa, joka pelasi valkoisilla kivillä voitti. Taranu pelasi uudestaan MoGo-ohjelmaa vastaan vuonna 2008 [Computer Go, 2014]. Nyt Taranu pelasi 4 peliä, joista hän hävisi viimeisen, kun MoGo-ohjelman ajansäästötekniikoita muutettiin niin, että se keskittyy enemmän pelin keskivaiheeseen alun sijasta [Computer Go, 2014].

MoGo-ohjelman versio 4.86 voitti 5 danin ammattilaisen Javier-Aleksi Savolaisen vuonna 2009. Myös MoGoTW voitti 9 danin ammattilaisen Chun-Hsun Choun pelissä, jossa molemmilla oli miettimisaikaa 45 minuuttia. Kun aikaa lyhennettiin 10 ja 15 minuutiin, niin MoGoTW-ohjelma hävisi. [Computer Go, 2014]



Kuva 10: Fuegon voitto Zhou Junxunia vastaan vuonna 2009
[Gelly et al., 2012]

Vaikka korkeimman tason ammattilainen on pystytty voittamaan nykyisillä go-ohjelmilla 9x9-laudalla, se ei kuitenkaan tarkoita sitä, että 9x9-lauta olisi lähellekään ratkaistu. 9x9-lautakoko on usein vaikeampi ammattilaispelaajille, jotka usein keskittyvät pelaamaan 19x19-kokoisilla laudoilla. Lautojen suuri kokoero vaikuttaa paljon siihen, kuinka peliä kannattaa pelata, ja 9x9-laudalla ammattilaispelaajat eivät välttämättä pysty soveltamaan heidän normaaleja strategioita.

Go-peliä on tarkoitus pelata 19x19-laudalla, vaikka muutkin laudat ovat mahdollisia. Taulukossa 2 on listattu joitain voittoja tietokone-go-ohjelmille 19x19-laudalle. Taulukon kaksi ensimmäistä riviä on listattu, jotta voidaan nähdä, millainen parannus tietokone-go-ohjelmissa on

tapahtunut. Taulukkoon on kerätty vain osa voitoista, mutta pelejä on pelattu huomattavasti enemmän, joista suurimman osan tietokoneet ovat hävinneet. Usein myös ihmispelaajat ovat pelanneet useamman pelin tietokoneohjelmaa vastaan, mutta taulukkoon on merkitty vain voitot.

Vuonna 1991 Goliath-ohjelma pystyi voittamaan vain 5 harrastelijadanin tasoisia pelaajia ja silloinkin Goliath tarvitsi tasoitukseksi hurjat 17 kiveä. Vuonnan 1995 Handtalk oli edelleen samalla tasolla, mutta tasoituskivien määrää oli pystytty vähentämään kolmeentoista. Vielä 1990-luvulla vastustajana oli vain nuorten mestareita ja Handtalk-ohjelman vastustajat olivat vain 9- ja 10-vuotiaita mestareita. [Computer Go, 2014]

Vuonna 2008 MoGo-ohjelma yhtäkkiä pystyi voittamaan 8 danin ammattilaisen Kim Myungwanin [Gelly et al., 2012; Computer Go, 2014]. MoGo-ohjelman eduksi oli annettu tasoitusta 9 kiveä [Gelly et al., 2012; Computer Go, 2014]. Tämä suuri harppaus go-ohjelmissa saatiin aikaan MCTS-algoritmin avulla. MoGo toimi myös Hyugensin supertietokoneella, jonka nopeus on noin tuhat kertaa nopeampi kuin IBM:n Deep Blue -ohjelmalla, joka pystyi vuonna 1997 voittamaan shakin maailmanmestarin [Kroeker, 2011]. Samana vuonna Crazy Stone voitti kaksi peliä Aoba Kaoria vastaan (4 dan), 7 ja 8 tasoituskivellä [Gelly et al., 2012].

Vuonna 2009 Many Faces of Go -ohjelma voitti yhden danin ammattilaisen James Kerwinin [Marcolino and Matsubara, 2010]. Many Faces of Go -ohjelma sai tasoitusta 7 kiveä, ja tämän voiton voi melkein kokea tason laskuksi, koska Kerwin ei ole kuin 1 danin ammattilainen ja kuitenkin tarvittiin saman verran tasoitusta kuin 4 danin ammattilaista vastaan.

Vuonna 2010 Zen-ohjelma voitti Aoba Kaorin kuudella tasoituskivellä [Gelly et al., 2012]. Zen pystyi parantamaan suoritustaan vuoteen 2012 mennessä, jolloin se voitti kaksi 9 danin ammattilaista Masaki Takemiyan ja Chun-Hsun Choun, vain neljän kiven tasoituksella. Vuonna 2013 myös CrazyStone onnistui voittamaan yhden pelin Ishida Yoshiota vastaan, hänkin tasoltaan 9 danin ammattilainen, 4 tasoituskiven avulla.

Vuosi	Ohjelma	Vastustaja	Vastustajan taso	Tasoitus
1991	Goliath	Kolme nuorta vastustajaa	5 harrastelija dan	17 kiveä
1995	Handtalk	Nuorten mestareita (9 ja 10 v)	5 harrastelija dan	15 ja 13 kiveä
2008	MoGo	Kim Myungwan	8 dan	9 kiveä
2008	Crazy Stone	Aoba Kaori	4 dan	8 kiveä
2008	Crazy Stone	Aoba Kaori	4 dan	7 kiveä
2009	Many Faces of Go	James Kerwin	1 dan	7 kiveä
2010	Zen	Aoba Kaori	4 dan	6 kiveä
2012	Zen	Masaki Takemiya	9 dan	4 ja 5 kiveä
2012	Zen	Chun-Hsun Chou	9 dan	4 kiveä
2013	Crazy Stone	Ishida Yoshio	9 dan	4 kiveä

Taulukko 2: 19x19-laudalla pelattuja voittoja tietokoneille

4. Monte Carlo -puuhaku

Monte Carlo -puuhakua (MCTS) käyttävät tämän hetken menestyneimmät ohjelmat, kuten Crazy Stone, Zen ja Fuego [Sensei's Library, 2014]. MCTS-menetelmä on yhdistelmä pelipuuta ja Monte Carlo -simulaatiota [Gelly et al., 2012; Hashimoto et al., 2011]. Sitä soveltavat ohjelmat saavat vain minimaalisen tietämyksen pelistä ennen kuin ne alkavat pelaamaan [Gelly et al., 2012]. Ne saavat asiantuntemuksensa pelaamalla simuloituja satunnaisia pelejä itsenäisesti [Gelly et al., 2012]. Algoritmia kutsutaan Monte Carlo -puuhauksi, koska se rakentaa ja laajentaa hakupuuta samalla, kun se arvioi yksittäisten siirtojen vahvuutta niiden menestyksen perusteella satunnaisessa pelissä käyttäen Monte Carlo -simulaatiota [Gelly et al., 2012].

Tässä luvussa aluksi esitellää pelipuun ja Monte Carlo -simulaation käyttäminen go-pelissä. Sen jälkeen esitellään Monte Carlo -puuhaku go-pelin yhteydessä ja pohditaan, mitä ongelmia MCTS-menetelmän käytössä vielä on. Seuraavaksi esitellään yleisin MCTS-menetelmän muoto, UCT-algoritmi, jota käytetään melkein kaikissa tietokone-go-ohjelmissa.

4.1. Pelipuun ja Monte Carlo -simulaation käyttö go-pelissä

Pelipuun tai Monte Carlo -simulaation käyttäminen go-pelissä ei tuota kovin hyviä tuloksia. Varsinkaan pelkkä pelipuu ei pysty löytämään hyviä vaihtoehtoja seuraaviksi siirroiksi. Monte Carlo -simulaatiota ei edes paljoa yksikseen käyetty ennen kuin keksittiin muokata Monte Carlo -puuhaku.

Täydellinen pelipuu sisältää täydellisen pelin. Go-pelissä täydellistä pelipuuta ei voida luoda, koska se vaatii liikaa muistia [Gelly et al., 2012]. Tämän takia pelipuu on luotu vain osittain ja minimax-haun on käyttäminen on hankalaa ja tuottamatonta. Pelipuun suuruuden takia voidaan tehdä myös alipuu, joka on syvyys-rajoitettu pelipuu [Gelly et al., 2012]. Alipuun lehdet on korvattu heuristisella arviointifunktiolla, koska niiden oikeita arvoja ei voida tietää [Gelly et al., 2012]. Arviointifunktio luodaan joko ihmiseksperttien avulla tai peleistä saadun tiedon kautta [Gelly et al., 2012]. Liun ja muiden [2008] mukaan pelipuu on kuitenkin melkein hyödytön tietokone-gossa, koska go-pelin hakuavaruus on paljon suurempi kuin muissa peleissä, eikä pelilaudan tilan tarkka ja staattinen arviointi ole helposti käsiteltävä.

Monte Carlo -simulaatio pelaa useita pelisimulaatioita, joiden avulla se hankkii mahdollisimman paljon tietoa [Niekerk et al., 2012]. Pelisimulaatiot pelataan aina tällä hetkellä olevasta pelitilanteesta pelin loppuun asti. Simulaatioiden avulla pystytään hankkimaan tärkeää tietoa useiden siirtojen laadusta [Niekerk et al., 2012].

Monte Carlo -simulaatio käyttää kahta simulaation menettelytapaa (policy) arvioimaan tilojen vahvuutta go-pelissä [Gelly et al., 2012]. Molemmille pelaajille luodaan oma simulaatiomenettelytapa. Tämä menettelytapa kertoo, kuinka ohjelman tulee pelata

pelisimulaatioita itsekseen, jotta se saa luotua arvoja eri siirroille [Gelly et al., 2012]. Monte Carlo -simulaatio on hyvin riippuvainen menettelytavan laadusta, joten mahdolliset ongelmat on korjattu satunnaisuuden avulla [Gelly et al., 2012], jolloin hyvistä siirroista valitaan satunnaisesti yksi. Satunnaisuudesta on kaksi hyötyä: se saattaa vähentää toistuvia virheitä, ja se auttaa tekemään eron helposti ja vaikeasti voitettavien tilojen välille [Gelly et al., 2012]. Tämä vastaa oikean ihmispelaajan suoritusta, koska sekään ei ole koskaan täydellinen [Gelly et al., 2012].

Vuonna 1993 ilmestyi Gobble, joka oli ensimmäinen tietokone-go-ohjelma, joka käytti hyväkseen Monte Carlo -simulaatiota. Gobble simuloi useita pelejä nykyisestä pelitilanteesta ja se yhdisti kaksi aivan uutta ideaa Monte Carlo -menetelmään: kaikki-liikkeet-niin-kuin-ensimmäinen (all-moves-as-first) heuristiikkaa (AMAF) ja järjestettyä simulaatiota. AMAF olettaa, että siirron arvo ei muutu huomattavasti vaikka pelitilanne muuttuu. Gobble myös järjesti kaikki siirrot niiden arvioitujen arvojen mukaan. Kaikki simulaatiot käyttivät sitten näitä järjestettyjä siirtoja. [Gelly and Silver, 2011]

4.2. Monte Carlo -puuhaku

MCTS-menetelmä on tällä hetkellä paras menetelmä tietokone-gossa. Se ei kuitenkaan toimi tarpeeksi hyvin ja siksi sitä on yritetty parantaa monin erin tavoin vaihtelevilla menestyksellä. MCTS-algoritmin onnistumisia ei täysin ymmärretä, eikä yhteismielisyyteen päästä edes siinä, mikä on sen suurin heikkous.

Perusmuodossaan MCTS ei pysty löytämää järkeviä siirtoja kohtuullisessa ajassa, koska laillisten siirtojen määrä on niin suuri, ettei avainsolmuissa välttämättä vierailla tarpeeksi, jotta niille saataisiin aikaan oikeanlaiset arvot [MCTS.ai, 2014]. Lisäksi tarvittavien simulaatioiden määrä voi olla jopa miljoonia, jotta löydetään parhaat mahdolliset siirrot [MCTS.ai, 2014]. Simulaatioiden määrä vie siis liikaa aikaa.

MCTS-menetelmän neljä eri vaihetta pitää osata ohjelmoida toimimaan yhteensopiviksi go-pelin kanssa, jos halutaan saada hyviä tuloksia MCTS-menetelmällä. MCTS-menetelmän neljä eri vaihetta valinta, laajentuminen, simulaatio ja takaisin eteneminen voidaan kaikki tehdä monella eri tavalla.

Valintavaiheessa pitää osata tasapainottaa etsimistä ja tutkimista [Chaslot, 2010]. Pitäisi valita siirto, joka johtaa parhaaseen tulokseen, mutta go-pelissä on hankala tietää, mitkä syyt johtavat parhaaseen lopputulokseen. Siksi on luotu useita MCTS-valintastrategioita, kuten OMC, PPBM, UCT ja UCB1-TUNED. OMC (Objective Monte Carlo) sisältää kaksi funktiota: kiirellisyys- ja oikeudenmukaisuusfunktio. Kiireellisyysfunktio päättää, kuinka kiirellinen siirto on ja oikeudenmukaisuusfunktio päättää, mikä siirto pelataan niiden kiirellisyyden perusteella. PBBM (probability to be better than best move) toimii muuten samalla lailla kuin OMC, mutta parhaan siirron keskihajonta otetaan huomioon. UCB1-TUNED on muutettu versio UCT-algoritmista, joista UCT-algoritmi esitellään seuraavassa luvussa. [Chaslot, 2010]

Laajentumisvaiheessa lisätään uusi solmu pelipuuhun. Joissain muissa peleissä tässä vaiheessa pystytään lisäämään useita solmuja tai jopa kaikki mahdolliset siirrot, mutta tietokone-gon tapauksessa suosittu strategia on lisätä yksi solmu per simuloitu peli [Chaslot, 2010].

Simulaatiovaiheessa ohjelma pelaa itse pelin, kunnes se saavuttaa pelin lopun. Tämä voi tapahtua pelaamalla satunnaisia siirtoja, mutta parempi strategia on käyttää pseudosatunnaisia siirtoja, jotka valitaan simulointistrategian avulla. Simulaatiota mietittäessä tulee päättää, haluaako panostaa hakuun vai tietämykseen. Tietämyksen lisääminen simulointeihin parantaa pelaamisvahvuutta, sekä tekee tuloksista luotettavampia ja tarkempia, mutta liian monimutkainen heuristinen tietämys on tietokoneelle raskaampi ja simulaatioiden määrä per sekunti pienenee huomattavasti. Monimutkainen simulaatiostrategia myös tekee simuloidusta pelipuusta lyhyemmän ja näin pelaamisen taso laskee, koska ei ehditä tutkia tarpeeksi eri haaroja. Toinen ongelma simulaatiostrategian teossa on se, että jos se on täysin satunnainen, niin silloin tutkitaan liikaa myös huonoja siirtoja, sekä simuloitu peli on silloin epärealistinen, jolloin koko ohjelman taso laskee. Jos simulointistrategia ei ole satunnainen, niin usein käy niin, että samassa solmussa vieraillaan liian usein ja tälloin hakuavaruutta tutkitaan liian vähän, mikä myöskin johtaa koko ohjelman tason huononemiseen. [Chaslot, 2010]

Takaisin etenemisvaiheessa edetään lehtissolmusta takaisin ylöspäin pelipuuta ja samalla lisätään solmuihin simulaation tulokset. Kaksinpeleissä kuten go-pelissä voitto on positiivinen, häviö on negatiivinen ja tasapelin sattuessa käytetään nollaa. Tulos lasketaan takaisin etenemisstrategialla, joita on keskiarvo, Max, tietoihin perustuva keskiarvo, Mix ja MCTS-ratkaisija. Keskiarvo on käytetyin ja tehokkain strategia, joka lasketaan laskemalla solmun lapsien simulaatioiden tuloksien keskiarvo. Muut strategiat ovat yrityksiä löytää parempi takaisin etenemisstrategia, johon ei vielä kuitenkaan olla pystytty. [Chaslot, 2010]

Kun kaikki vaiheet on käyty läpi moneen otteeseen, niin tulee siirron valitseminen. Tämänkin pystyy tekemään monella eri tapaa. Siirto on kuitenkin paras lapsisolmu, joka juurella on. Parhaan lapsisolmun voi määritellä monella tapaa, mutta Chaslotin [2010] mukaan, jos simulaatioita on tehty useita, niin ei ole merkitystä, miten lapsisolmun valitsee. Mutta jos simulaatioita on vain vähän, niin valitsemalla suurimman arvon omaavan lapsisolmun oli huonompi kuin muut tavat. Nämä muut tavat ovat vakaa lapsi (robust child), vakaa max lapsi sekä varma lapsi (secure child). Vakaa lapsisolmu on se solmu, joissa on vierailtu useimmiten. Vakaa max lapsisolmu on sellainen solmu, jolla on eniten vierailuja ja sen arvo on suurin (jos tällaista solmua ei ole jatketaan simulaatioita niin kauan kuin sellainen on). Varma lapsisolmu on sellainen, joka maksimoi alemman luottamusrajan. Eli se on solmu, joka maksimoi funktion $Solmun\ arvo + \frac{Parametri}{\sqrt{vierailumäärä}}$ [Chaslot, 2010]

MoGo käyttää valintavaiheessa käsintehtyä menettelytapaa ja simulaatiovaiheessa se käyttää yksinkertaisia sääntöjä massiivisen kuvuokirjaston sijasta, jota useat muut tietokone-go-ohjelmat

käyttävät. MoGo:n säännöt toimivat minkä tahansa vastustajan siirron jälkeen näin [Silver and Gelly, 2011]:

1. Jos vastustajan siirto asettaa kiviä atariin, pelaa satunnainen pelastava siirto.
2. Jos jokin kahdeksasta risteyksestä, jotka ympäröivät vastustajan siirtoa, on yksinkertainen leikkaava kuvio tai hane, pelaa silloin satunnaisesti jompikumpi.
3. Jos jokin vastustajan kivistä voidaan valloittaa, pelaa valloittava siirto satunnaisesti.
4. Muuten pelaa satunnainen siirto.

Hashimoto ja muut [2011] ovat sitä mieltä, että MCTS-algoritmin heikkous on sen pelipuun petollinen rakenne. Go-pelille ei pystytä rakentamaan täydellistä pelipuuta, koska puu on liian valtava, siksi käytetään vain osittaista puuta. Sama ongelma on pelkän pelipuun käytössä. Ongelmaa yritetään hoitaa lisäämällä pelitietämystä, yleistämällä alipuita sekä säätämällä simulaation menettelytapaa ”käsin” [Hashimoto et al., 2011]. Pelitietämyksen lisääminen on todella vaikeaa pelin suuren vaihtelevuuden takia. Jos ongelmat pystyttäisiin oikeasti korjaamaan pelitietämyksellä ja simulaation menettelytapojen säätämisellä, olisi go-peli pystytty jo ajat sitten ratkaisemaan, koska tätä on yritetty jo melkein heti ensimmäisestä tietokone-go-ohjelmasta lähtien. Rimmel ja muut [2010] ovat myös sitä mieltä, että MCTS-algoritmin suurin heikkous on simulaation menettelytavan valitseminen ja luominen. Silverin ja muiden [2012] mielestä MCTS-algoritmilla on kaksi ongelmaa: ensimmäinen on se, että kaikki tilat arvioidaan itsenäisesti ilman yleistystä samanlaisista aiemmista tiloista; toinen ongelma on se, että Monte Carlo -simulaatio tuottaa korkeita muuttujan arvioita jokaisen tilan arvosta. Eli se ei anna siirroille tarpeeksi vaihtelevia arvoja niiden hyötyyn nähden. Bourki ja muut [2010] väittävät, että MCTS-algoritmi on erittäin heikko semeai-pelitilannetta vastaan. Semeai on kilpajuoksu pelaajien välillä, kumpi saa valloitettua alueen [Müller, 2002].

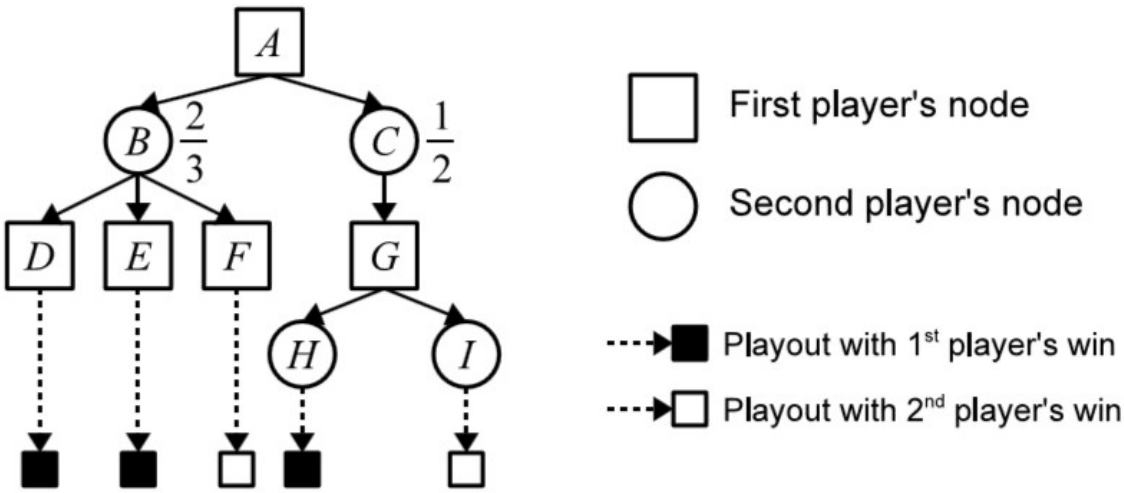
4.3. UCT-algoritmi

UCT-algoritmia kehittivät Kocsis ja Szepesvári vuonna 2006, ja sitä käytetään MCTS-menetelmän valintavaiheessa valitsemaan solmuja pelipuusta [Chaslot, 2010; Sensei's Library, 2014]. UCT-algoritmi on yleinen monissa go-peliä pelaavissa MCTS-algoritmia toteuttavissa ohjelmissa. Muun muuassa MoGo-, CrazyStone-, Pachi-, Fuego- ja Zen-ohjelmat käyttävät UCT-algoritmia [Sensei's Library, 2014]. UCT-algoritmi (Upper confidence bound applied to trees) eli luottamusyläraja sovellettuna puihin perustuu monikätinen rosvo (multi-armed bandid) -ongelmaan [Marcolino and Matsubara, 2011; Gelly et al., 2012]. UCT-algoritmi on myös yleisnimitys MCTS-menetelmälle, joka käyttää UCT-valintastrategiaa.

UCT-algoritmilla yritetään ratkaista monikätinen rosvo -ongelmaa. MCTS-menetelmän toiminta on stokastista ja epätäydellistä, joten pelipuun arvoissa on luontaista satunnaisuutta [Gelly

et al., 2012]. Siksi on vaikea arvioida, milloin tulisi valita optimaalinen ja milloin suboptimaalinnen siirto. Tätä ongelmaa kutsutaan monikätinen rosvo -ongelmaksi. Kun siirtoa valitaan, tulee pohtia, valitaanko paras siirto tällä hetkellä vai tutkitaanko muita siirtoja, jotta nähdään, toimisivatko ne paremmin [Cazenava and Jouandeu, 2008]. Go-pelin laudalla jokainen tilanne nähdään rosvona ja jokainen siirto nähdään kätenä, jossa on tuntematon palkinto [Marcolino and Matsubara, 2011].

UCT-algoritmi valitsee siirron pelipuusta luottamusyläraja-arvon (upper confidence bound) avulla [Hashimoto et al., 2011]. Jokaisessa sisäisessä solmussa n UCT-algoritmi valitsee siirron j, jossa on suurin luottamusylärajan arvo määritelty kaavassa $ucb_j := r_j + C\sqrt{\frac{\log s}{n}}$ [Hashimoto et al., 2011]. Kuvassa 11 on havainnollistettu UCT-algoritmin toimintaa. Kuvan hetkellä UCT-algoritmi valitsisi B-siirron, koska se tuottaa voiton kaksi kolmesta kerrasta. C-siirron voittoprosentti on vain puolet ja näin se on huonompi vaihtoehto.



Kuva 11: UCT-algoritmi [Hashimoto et al.,2011].

Hashimoton ja muiden [2011] mielestä siirron löytäminen vie liian kauan aikaa UCT-algoritmilla, koska algoritmi käyttäytyy hyvin optimistisesti. Gelly ja muut [2012] ovat sitä mieltä, että UCT-algoritmi pystyy löytämään parhaan siirron, jos sille antaa tarpeeksi aikaa. Hashimoto ja muut [2011] sanovat, ettei yleensä voida antaa tarpeeksi aikaa, jotta UCT-algoritmi ehtisi löytämään parhaan ratkaisun. UCT toimii parhaiten, kun suuren alipuun lehdet sisältävät samanlaiset tulokset [Gelly et al., 2012]. Gelly ja muut [2012] ovat sitä mieltä, että vaikka UCT-algoritmissa on puutteita, kannattaa se silti lisätä go-peliä pelaaviin ohjelmiiin, koska se on yksinkertainen lisätä ohjelmaan, sillä on hyvä laajennettavuus ja sen empiirinen menestys on ollut hyvä monilla eri aloilla.

Hashimoto ja muut [2011] kehittivät kiihdytetyn UCT-algoritmin, jolla yritetään nopeuttaa hakua pelipuusta ja välttää ongelmia, jotka johtuvat siitä, että ei pystytä käyttämään kokonaista pelipuuta. Kiihdytetty UCT-algoritmi toimii uudella varaoperaattorilla ja se painottaa uusia käytettyjä siirtoja ja antaa pienemmän arvon vanhemmille siirroille. Uudet pelisimulaatiot ovat arvokkaampia kuin vanhat [Hashimoto et al., 2011]. Hashimoton ja muiden [2011] mukaan kiihdytetty UCT-algoritmi paransi Fuegon vahvuutta. Tämä tulos on kuitenkin saatu pelaamalla peli, jossa vastakkain on kiihdytetty UCT-algoritmi lisättynä Fuegoon vastaan Fuego. Mielestäni ei voida sanoa varmasti, että algoritmi parantaa Fuegon vahvuutta, koska algoritmi saattoi vain löytää Fuegon heikkouden ja toimivan hyvin Fuegoa vastaan.

5. MCTS-laajennukset

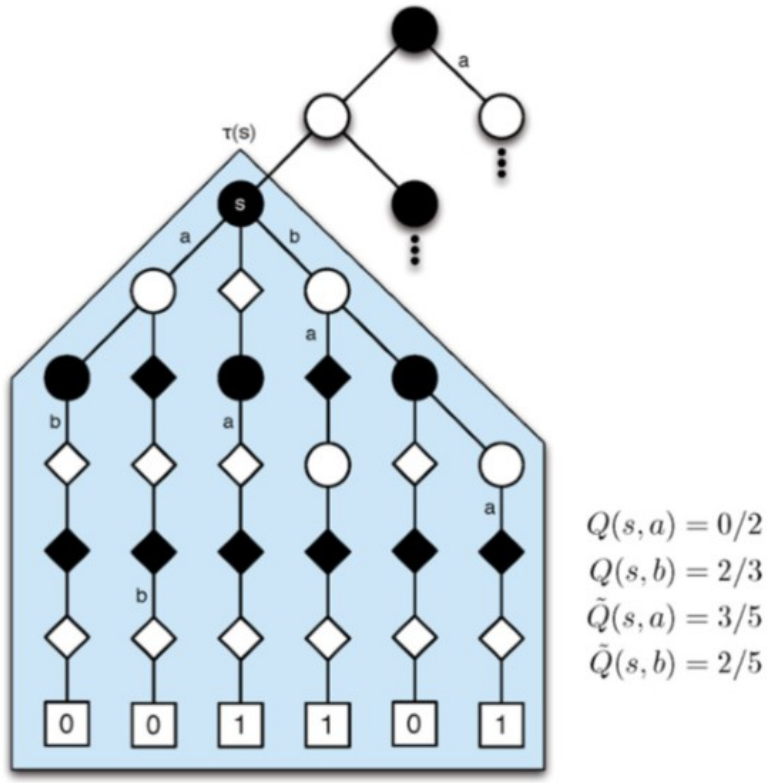
MCTS-laajennuksia on lisäilty tietokone-gohon melkein heti MCTS-menetelmän julkaisemisen jälkeen. Tarkoituksena on tehostaa MCTS-menetelmän toimintaa ja korjata sen mahdollisia virheitä. Laajennukset voidaan jakaa kahteen luokkaan: tietämykseen perustuvat sekä itsenäisesti toimivat [MCTS.ai, 2014]. Tietämykseen perustuvat lisäykset ovat sellaisia, joihin MCTS-menetelmään on lisätty esimerkiksi tietokanta, jonka avulla hyviä siirtoja etsitään; tämä voi huomattavasti parantaa tuloksia, mutta nopeuden ja yhtenäisyyden kustannuksella [MCTS.ai, 2014]. Itsenäisesti toimivat laajennukset ylläpitävät ohjelman yhtenäisyyttä ja parannukset yleensä lisätään pelipuuhun tai simulaatioihin [MCTS.ai, 2014]. Tässä luvussa esitellään muutama MCTS-laajennus, joista osa löytyy jo melkein jokaisesta tietokone-go-ohjelmasta ja osa on vasta aivan luomis- ja tutkimusvaiheessa. Ensimmäisenä esitellään RAVE, joka parantaa UCT-algoritmin suorituskykyä. Seuraavaksi syvennytään rinnakkaislaskentaan, jonka käyttö on hyvin yleistä varsinkin tietokone-go-turnauksissa. Lopuksi tarkastellaan kahta hieman tuntemattomampaa tekniikkaa tietokone-gossa: solmulaajennusta ja meta-MCTS:ta.

5.1. RAVE

RAVE (rapid action value estimation) eli nopean toiminnan arvon arviointi on vahva heuristiikka, joka usein parantaa UCT-pohjaisten algoritmien suorituskykyä [Tom and Müller, 2010]. RAVEa käyttää mm. Fuego [Hashimoto et al., 2011]. RAVE on todettu menestyksekkääksi myös muissa peleissä kuten Hex ja Havannah [Tom and Müller, 2010]. Vaikka UCT on vahva algoritmi, Monte Carlo -puuhaun menestys on laajalti muiden parannuksien kuten RAVE:n ansiota [Tom and Müller, 2010].

RAVE yleistää AMAF-heuristiikan hakupuihin [Gelly et al., 2012]. AMAF-heuristiikka oletti, että siirtojen arvo ei muutu huomattavasti, vaikka pelitilanne muualla lautaa muuttuu. MCTS-menetelmä ei itsekseen pysty yleistämään samankaltaisia siirtoja ja pelitilanteita, koska se järjestää kaikki puurakenteeksi, jossa sama siirto voi esiintyä monesti, joita kuitenkin kohdellaan kuin eri siirtoja [Gelly and Silver, 2010]. MCTS joutuu aina jokaiseen pelitilanteeseen simuloimaan kaikki siiirrot, jotta paras siirto löydetään. RAVE-heuristiikalla halutaan löytää siirtojen väliset yhteenkuuluvuudet ja tilastoida sekä arvioida saman siirron hyöty kokonaisuudessaan. Siirron kaikki esiintymät etsitään puusta ja tutkitaan, kuinka moni simulaatio johti voittoon, jossa kyseistä siirtoa käytettiin [Gelly et al., 2012]. Tämä ei tietysti aina toimi oikein, koska on paljon sellaisia pelitilanteita, kuten taktisia taisteluita alueista, jolloin siirron arvo muuttuu huomattavasti muutoksista pelilaudalla, joskus tehden siirron turhaksi ja joskus paljon tärkeämmksi [Gelly and Silver, 2010].

Kuvassa 12 on esitetty, kuinka RAVE-algoritmi arvioi solmujen arvoja. Kuvassa tutkitaan pelitilanteen s simulaatioita, joita on tehty kuusi kappaletta. Merkinnät a ja b ovat siirtoja, jotka tulevat useasti simulaatioissa esille. Simulaatioiden lopputulokset on merkitty neliöruutuihin. Monte Carlo -simulaatio valitsisi siirron b, koska siirron a käyttäminen ensimmäisenä johti kahteen häviöön. RAVE taas suosii siirtoa a, koska sen käyttäminen johti voittoon kolme kertaa viidestä, eikä huomioida, milloin siirtoa käytettiin simulaatiossa. Siirto b ei johtanut voittoon kuin yhden kerrran kolmesta, joka on huonommin kuin siirrolla a. [Gelly and Silver, 2011]



Kuva 12: RAVE-algoritmin toiminta [Gelly and Silver, 2011]

RAVE ei ole aina luotettava. On tiedossa, että RAVE aiheuttaa satunnaisia virheitä suosimalla joskus virheellisiä siirtoja [Tom and Müller, 2010]. Esimerkiksi jos siirto on erittäin hyvä tällä hetkellä, mutta yleisesti huono, kun se pelataan myöhemmin simulaatiossa, niin siirron RAVE-arvo on pahasti aliarvioitu [Tom and Müller, 2010]. UCT-algoritmi ilman RAVE-heuristiikkaa saattaa löytää oikean siirron, kun RAVE epäonnistuu. Tomin ja Müllerin [2010] mielestä tulevaisuudessa tulee kehittää algoritmi, jonka avulla pystytään hyödyntämään keskiarvojen ja RAVE-arvojen eroja. Gellyn ja muiden [2012] mielestä RAVE:n hakema lisätieto on joskus harhaanjohtavaa. RAVE:a syytetään usein huonoista asymptoottisesta käyttäytymisestä joissakin MCTS-ohjelmissa, mutta Bourkin ja muiden [2010] mielestä RAVE:n poistaminen ei ratkaise ongelmaa, koska MCTS-menetelmä ei itsestään osaa myöskään ratkaista ongelmia.

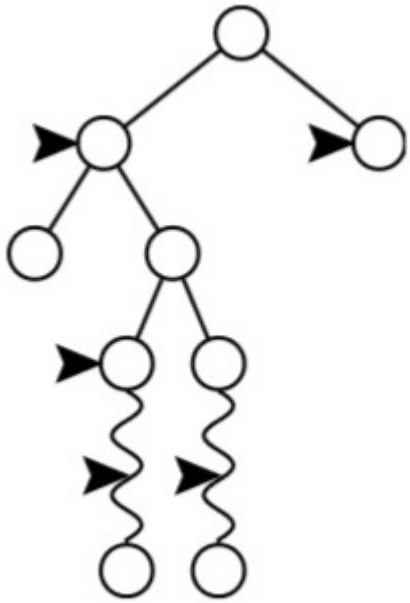
Rimmel ja muut [2010] tekivät oman poolRave-version RAVE:sta. Sen periaate on pelata siirtoja, jotka uskotaan tehokkaiksi RAVE-arvojen perusteella. RAVE-arvoja ovat solmut, joille on tehty ainakin 50 simulaatiota. Rimmel ja muut [2010] saivat 51,7% voittoprosentin muokatulle järjestelmälleen, kun he pelasivat muokkaamatonta järjestelmää vastaan. Kun he poistivat alkuperäisestä järjestelmästä go-pelin tiedot, tuli voittoprosentiksi 62,7%. Rimmelin ja muiden [2010] mielestä poolRaven lisääminen ohjelmaan kannattaa.

5.2. Rinnakkaistaminen

Rinnakkaistaminen tapahtuu jaetun muistin ja moniydinkoneen avulla [Gelly et al., 2012]. Ytimet jakavat tietoa säikeiden avulla ja erillisiä simulaatioita voidaan toteuttaa säikeissä samanaikaisesti [Gelly et al., 2012]. Rinnakkaistamisen hyöty on siinä, että sillä pystytään lisäämään pelisimulaatioiden määrää. Simulaatioiden lisäämiseen käytetään prosessoituja solmuja, jotka on joko jaettu eri koneille ryppäissä (cluster) tai yksi solmu on keskusyksikkö (CPU) symmetrisessä moniprosessointikoneessa [Nieker et al., 2012].

Klassiset pelipuuhaut, kuten minimax-haku, ovat vaikeita rinnakkaistaa, mutta MCTS-algoritmi on hieman helpompi rinnakkaistaa [Gelly et al., 2012]. MCTS-algoritmin kolmas vaihe, simulointi, on mahdollista toteuttaa rinnakkaislaskennalla niin, että simuloitavat peli voidaan pelata täysin itsenäisesti [Chaslot et al., 2008]. UCT-algoritmin rinnakkaistaminen on kuitenkin osoittautunut hankalaksi. MCTS-algoritmin pystyy rinnakkaistamaan kolmella eri tavalla: puurinnakkaistaminen, lehtirinnakkaistaminen ja juuririnnakkaistaminen. Kaikilla näillä rinnakkaistamistavoilla on sama ongelma: rinnakkaisvaikutus (parallel effect). Rinnakkaisvaikutus tarkoittaa ohjelman vahvuuden katoamista rinnakkaistamisen takia, joka tekee rinnakkaistamisen hyödyn tyhjäksi. [Niekerk et al., 2012]

Puurinnakkaistamisen esittelivät Chaslot ja muut [2008], ja nykään se on yleisin rinnakkaislaskentatekniikka go-pelin yhteydessä [Segal, 2010]. Puurinnakkaistamisessa on yksi jaettu MCTS-puu, jota kaikki prosessointisolmut käyttävät [Niekerk et al., 2012]. Myös muisti on jaettu, ja se toimii moniydinsysteemissä [Niekerk et al., 2012]. Kuvassa 13 on esitetty, kuinka puurinnakkaistaminen toimii; nuolet kuvaavat prosessointisolmuja, jotka työskentelevät jaetussa MCTS-puussa yhtäaikaisesti [Niekerk et al., 2012]. Puurinnakkaistamisen ongelmana on liiallinen etsintä (over exploration), joka tarkoittaa sitä, että samaa työtä tehdään monessa eri prosessissa turhaan [Niekerk et al., 2012]. Chaslot ja muut [2008] yrittivät parantaa puurinnakkaistamista kahdella lisäyksellä: virtuaalinen tappio (virtual loss) -tekniikalla ja käyttämällä paikallista keskinäistä poissulkevuutta (local mutex). Paikallinen keskinäinen poissulkevuus -menetelmässä rinnakkaiset säikeet käyttävät poissulkevuusmekanismia, jolla estetään tiedon korruptio niin, että muita säikeitä ei päästetä samaan haaraan samanaikaisesti [Chaslot, 2010; Chaslot et al., 2008]. He kuitenkin huomasivat, ettei keskinäinen poissulkevuus toiminut hyvin, eikä sitä kannata käyttää.

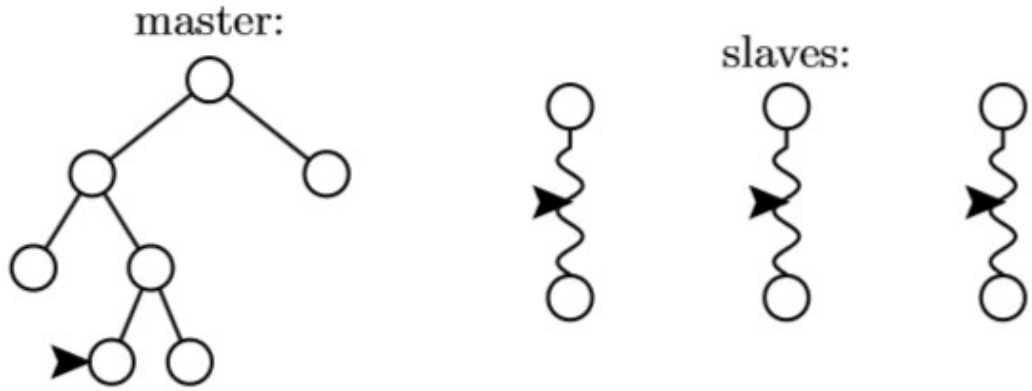


Kuva 13: Puurinnakkaistaminen [Niekerk et al., 2012].

Virtuaalinen tappio -tekniikka on luotu poistamaan puurinnakkaistamisen liiallista etsintää [Niekerk et al., 2012]. Tekniikka toimii niin, että kun säie kulkee pelipuuta pitkin valinta-vaiheessa, lisätään vierailtuihin solmuihin virtuaalinen tappio [Chaslot et al., 2008]. Solmun arvoa siis lasketaan ja silloin seuraava säie valitsee solmun vain silloin, jos se on edelleen parempi kuin solmun sisarukset (joille ei ole vielä annettu virtuaalista menetystä) [Chaslot et al., 2008]. Näin poistetaan ongelma, että säikeet tekevät samaa työtä turhaan [Niekerk et al., 2012]. Virtuaalinen tappio poistetaan takaisin eteneminen -vaiheessa, ja virtuaalisen tappion lisännyt myös poistaa sen [Chaslot et al., 2008]. Virtuaalinen tappio -mekanismin avulla löydetään solmut, jotka ovat selvästi parempia kuin muut, ja nämä solmut tutkitaan kaikilla säikeillä [Chaslot et al., 2008]. Solmut, joilla on epävarmat ja huonot arvot, tutkitaan vain kerran [Chaslot et al., 2008]. Virtuaalinen tappio -tekniikan käyttö on tärkeää ainakin silloin, kun prosessorien määrä on suuri [Chaslot et al., 2008].

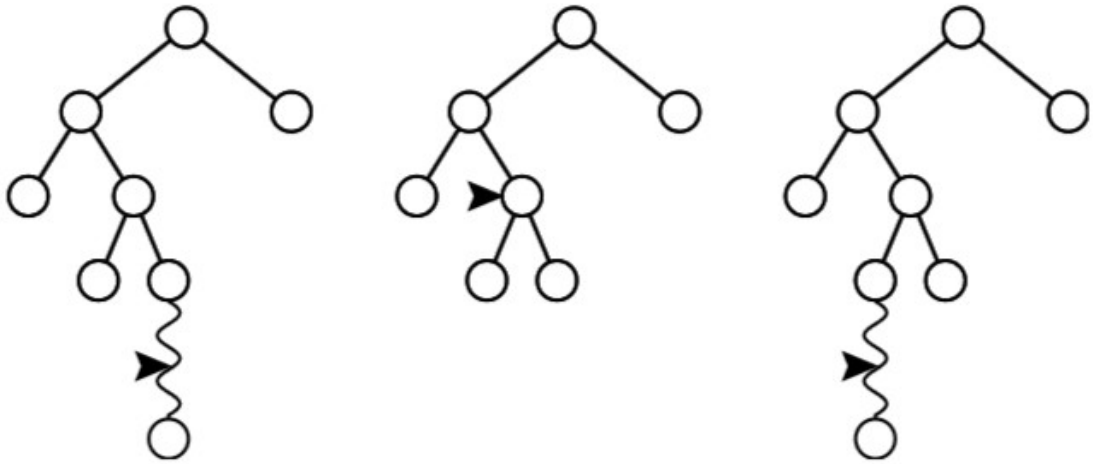
Lehtirinnakkaistaminen on helpoimpia tapoja rinnakkaistaa MCTS-algoritmi [Chaslot et al., 2008]. Lehtirinnakkaistaminen toimii yhdella isäntäsolmulla, joka komentaa useita orjasolmuja. Orjasolmujen tehtävä on pelien simulointi ja isäntäsolmu tekee kaiken muun. Se ylläpitää MCTS-puuta ja pyytää orjia simuloimaan tietystä lehdestä. Isäntäsolmu kerää tiedon orjilta ja päivittää MCTS-puun. Kuvassa 14 on isäntäsolmu, joka toimii MCTS-puuna, sekä orjasolmut, jotka toteuttavat pelisimulaatioita isäntäsolmulle. Lehtirinnakkaistamisen ongelma on se, että isäntäsolmu voi muuttua pullonkaulaksi. [Niekerk et al., 2012] Lehtirinnakkaistamisessa simuloidun pelin kesto on ennalta-arvaamaton [Chaslot et al., 2008]. Chaslotin ja muiden [2008] mielestä pelkkä

lehtirinnakkaistaminen on huono tapa rinnakkaistaa, koska muut rinnakkaistamistavat toimivat paremmin go-pelin yhteydessä.



Kuva 14: Lehtirinnakkaistaminen [Niekerk et al., 2012].

Juuririnnakkaistamista kutsutaan myös nimellä yksiajorinnakkaistaminen (single-run) [Chaslot et al., 2008]. Juuririnnakkaistaminen toimii niin, että jokaisella solmulla on oma MCTS-puu, ja ne toteuttavat kaikki neljä vaihetta MCTS-algoritmista [Niekerk et al., 2012]. Solmut jakavat keskenään tietoja, mutta ei kaikkea kerralla, vaan vain osia omista puistaan [Niekerk et al., 2012]. Kuvassa 15 näkyy, kuinka solmujen omat MCTS-puut tekevät pelisimulaatioita itsekseen [Niekerk et al., 2012]. Chaslot ja muiden [2008] mukaan juuririnnakkaistaminen vaikuttaa pätevältä tavalta rinnakkaistaa MCTS-ohjelma, koska se on yksinkertainen ja tehokas. Heidän testauksissaan 13x13-laudalla parhaat tulokset saatiin juuririnnakkaistamisella. Niekerkin ja muiden [2012] mielestä juuririnnakkaistaminen olisi yhtä hyvä kuin puurinnakkaistaminen, jos se pystyisi päivittämään solmuja äärettömällä taajuudella, jakamaan koko puu kerralla ja pelipuun päivitys tapahtuisi nolla-ajassa. Tämä kuulostaa jo teoriassa mahdottomalta.



Kuva 15: Juuririnnakkaistaminen [Niekerk et al., 2012].

Niekerk ja muut [2012] tutkivat rinnakkaistamista muuttamalla MCTS-pohjaista ohjelmaa Oakfoam tukemaan moniydin- ja rykelmärinnakkaistamista. He testasivat moniydinrinnakkaistamista käyttäen puurinnakkaistamistekniikkaa. Tällöin prosessoivat solmut ovat keskusyksikköytimiä moniydinsysteemissä. MCTS-puu on jaettu kaikkien prosessoivien solmujen kesken ja puu on osa jaettua muistia. Kaikki suoritettavat säikeet toimivat samassa puussa. Niekerk ja muut [2012] testasivat moniydinrinnakkaistamista kahdeksaan ytimeen asti ja virtuaalinen menetys -lisäyksellä. He eivät huomanneet negatiivista rinnakkaisvaikutusta 9x9- tai 19x19-laudoilla. Heidän mukaansa moniydinrinnakkaistaminen saavutti melkein ihanteellisen skaalauksen heidän testaamassaan laitteistossa. Rykelmärinnakkaistamisessa lehti- ja juuririnnakkaistaminen ovat ainoita realistisia vaihtoehtoja, joten Niekerk ja muut [2012] valitsivat juuririnnakkaistamisen. Silloin jokainen prosessoiva solmu jakaa osan omasta pelipuustaan muiden solmujen kanssa. He käyttivät solmujen väliseen viestintään viestinvälitysrajapintaa (MPI) ja testasivat rykelmärinnakkaistamista 8 ja 32 solmuun asti 9x9 ja 19x19-laudoilla. Pienemmällä 9x9-laudalla ohjelman vahvuus nousi hyvin vähän tai ei ollenkaan. Isomman laudan tapauksessa he huomasivat, että skaalaus 8 solmuun asti oli yhtä vahva kuin ideaalinen vahvuus olisi 4 solmulla. Niekerk ja muut [2012] totesivat, että liian monella solmulla ajaminen 19x19-laudalla on haitallista ohjelman vahvuudelle, koska 32 solmua toimi huonommin kuin 16 solmua. He jatkavat, että rykelmärinnakkaistamisessa on paljon parantamisen varaa varsinkin ideaalisen skaalauksen kannalta.

MCTS-algoritmin rinnakkaistaminen on osoittatunut hankalaksi. Segalin [2010] mielestä kaikista onnistunein on osittainen puusynkronointimalli, jota käyttää MOGO-TITAN. Segalin [2010] tulokset osoittavat, että MCTS-menetelmä pystyy skaalautumaan melkein täydellisesti ainakin 64 säikeeseen asti, kun se yhdistetään virtuaalisen menetykseen. Ilman virtuaalista menetys

-lisäystä skaalautuvuus on rajoittunut vain kahdeksaan säikeeseen [Segal, 2010]. Gellyn ja muiden [2012] mielestä MCTS-algoritmin rinnakkaistaminen ei tuo kovin suurta etua go-ohjelmiin, sillä MCTS pystyy kasvattamaan pelipuuta valinnaisesti aikaisempien simulaatioiden perusteella, mutta rinnakkaistetulla MCTS-algoritmilla täytyy sokeasti ajaa useita simulaatioita ilman, että otetaan huomioon samanaikaisia simulaatoita [Gelly et al., 2012]. Vaikka rinnakkaistamisen hyödyt saattavat jäädä minimaalisiksi, silti useimmat nykyiset tietokone-go-ohjelmat käyttävät rinnakkaistamista hyväkseen.

5.3. Solmulaajennus

Yajima ja muut [2010] tutkivat, kuinka solmulaajennus-operaattorit toimivat yhdessä UCT-algoritmin kanssa. He esittelivät kuusi operaattoria solmulaajennukseen (node-expansion): kaikki päät (all ends), vierailumäärä (visit count), sisarukset2 (siblings2), siirtymisen todennäköisyys (transition probability), merkittävä voittoprosentti (salient winnig rate) sekä vierailumäärän arviointi (visit count estimate) -laajennus. Yajiman ja muiden [2010] toteutuksessa laajennus on prosessi, jossa valittuun siirtoon lisätään uusi solmu hakupuuhun, joka kuvaa laudan tilaa siirron jälkeen. Uuteen solmuun lisätään myös kaikki mahdolliset siirrot, joita pelin tilassa on mahdollista seuraavaksi tehdä.

Kaikki päät -laajennusoperaattori toimii niin, että jokaisen iteraation jälkeen lisätään uusi solmu. Vierailumäärä-laajennusoperaattori odottaa, kunnes solmussa on vierailtu useammin kuin solmulla on sisaruksia. Vierailumäärä-laajennusoperaattorin voittoprosentti kaikki päät -laajennusoperaattoria vastaan oli 92,5% Yajiman ja muiden [2010] testauksissa. Kaikki päät operaattori oli myös hitain kaikista. Sisarukset2 toimi suurin piirtein yhtä hyvin kuin vierailumäärä. Sisarukset2 odottaa, kunnes solmussa on vierailtu kaksi kertaa useammin kuin solmun sisaruksissa. [Yajima et al., 2010]

Yajima ja muut [2010] loivat siirtymisen todennäköisyys, merkittävä voittoprosentti sekä vieralumäärän arviointi -laajennusoperaattorit. He testasivat näitä kolmea operaattoria vain vierailumäärä-operaattoria vastaan. Siirtymisen todennäköisyys -operaattori laajentaa solmun, jolla on korkea arviointi sen sisaruksiin nähden. Tämä operaattori toimii offline-tiedoilla ja on riippuvainen tietämyksestä. Merkittävä voittoprosentti odottaa, kunnes solmun voittoprosentti on paljon parempi kuin sisarsolmujen. Operaattori käyttää online-tietoa ja on siksi itsenäinen (domainista). Vierailumäärän arviointi -operaattori ei ole sopiva MCTS-algoritmille, joten sitä ei käsitellä tässä tutkielmassa. Sekä siirtymisen todennäköisyys ja merkittävä voittoprosentti suoriutuivat paremmin kuin vierailumäärä, ja parhaiten näytti pärjäävän merkittävä voittoprosentti.

Yajima ja muut [2010] ovat vahvasti sitä mieltä, että solmun laajennus on kannattava lisä UCT-algoritmiin. Heidän kokeelliset tulokset näyttivät, että kaikki solmulaajennusoperaattorit parantsivat huomattavasti UCT-algoritmin suorituskykyä.

5.4. Meta-MCTS

Meta-MCTS on yhdistelmä Monte Carlo -puuhakua ja sisäkkäistä Monte Carloa (nested Monte Carlo). Sisäkkäinen Monte Carlo yhdistää sisäkkäisiä kutsuja (nested calls) Monte Carlo -simulaation satunnaisuuden kanssa [Cazenava, 2009]. MCTS-algoritmin satunnainen menettelytapa on vaihdettu MCTS-menettelytapaan. Chou ja muut [2011] käyttivät meta-MCTS:ää luomaan go-pelin 7x7 kokoiselle laudalle aloituskirjan (opening book). Aloituskirja on kokoelma erilaisia hyviä aloitussiirtoja, jotka todennäköisesti pelataan pelin alussa. Aloituskirja päättää, minkä siirron pelaa ilman erillistä hakua [Chaslot, 2010]. Vaikka Meta-MCTS:n simulaatiot ovat melko hitaita, pelipuusta valitut siirrot ovat korkealaatuisia ja niistä sai hyvän aloituskirjan. Meta-MCTS laajennukset voidaan määritellä seuraavasti [Chou et al. 2011] :

1. MCTS on rakennettu käyttäen oletusarvoista (pseudosatunnaista) simulaatiostrategiaa
2. Meta-MCTS on rakennettu käyttämään MCTS-ohjelmaa simulaatiovaiheessa
3. Meta-Meta-MCTS on rakennettu käyttämään Meta-MCTS-ohjelmaa simulaatiovaiheessa
4. jne…

Tällaisia sisäkkäisiä tasoja on käytetty onnistuneesti Monte-Carlo -simulaatiossa. Chou ja muut [2011] kokeilivat, kuinka Meta-MCTS-menetelmän käyttö onnistuu go-pelissä käyttäen 7x7-kokoista pelilautaa. He rakensivat Meta-MCTS-menetelmän MoGoTW 4.86 Soissons -ohjelman päälle [Chou et al., 2011]. Meta-MCTS-menetelmän simulaatiovaihe on muutettu niin, että jo valmis MCTS-ohjelma MoGo suorittaa simulaatiot, eikä mikään satunnainen yms. simulaatiomenettelytapa [Chaslot, 2010].

Choun ja muiden [2011] Meta-MCTS käyttää kahta sääntöä, kun se valitsee siirtoja:

1. Tee siirto, jolla on suurin empiirinen onnistumisprosentti.
2. Tee siirto, jolla on suurin empiirinen onnistumisprosentti, jos onnistumisprosentti on suurempi tai yhtäsuuri kuin 10%. Muuten käytä oletusmenettelytapaa (Meta-MCTS tapauksessa oletusmenettelytapana on menettelytapa, jota käytetään MCTS:ssä).

Chou ja muut [2011] käyttivät Sensei's Library -nettisivua hyväkseen luodessaan Meta-MCTS-menetelmään ihmisasiantuntijuutta. He hyödynsivät Sensei's Library -nettisivua niin, että he käyttivät sivulla olevia aloituksia alustavana aloituskirjana sekä sivua harjoitusparina ohjelmalleen. Chou ja muut [2011] huomasivat, että Meta-MCTS oppi nopeasti minkä tahansa joukon muuttujia, jos sillä oli käytössä aloituskirja. Ilman aloituskirjaa ohjelman kyky oppia laski huomattavasti. Testit näyttivät, että Meta-MCTS-menetelmä oppi nopeasti pelaamaan uusia vastustajia vastaan. Se ei kuitenkaan pystynyt oppimaan ilman ulkoista tietämystä, kuten käyttämättä Sensei's Library -nettisivun tietämystä. [Chou et al., 2011]

Chou ja muut [2011] testasivat ohjelmaansa ihmispelaajia vastaan. Se voitti 20 peliä. Yhdessä pelissä heidän ohjelma teki virheen, joka olisi saattanut johtaa häviöön, mutta myös vastustajana ollut ihmispelaaja pelasi virheellisesti ja siksi ohjelma pystyi voittamaan. Pelaajat olivat ammattilaistasoisia. Voitot tapahtuivat kuitenkin vain 7x7-laudalla, jolla monet ammattilaispelaajat eivät ole tottuneet pelaamaan, eivätkä siksi pelaa niin hyvin kuin esimerkiksi 19x19-laudalla. Tämä lautakoko valittiin siksi, koska se on seuraava lauta, jota ei ole vielä ratkaistu [Chou et al., 2011]. Chou ja muut käyttivät komi-kokoa 9,5 valkoisella pelaajalla ja 8,5 mustalla pelaajalla. He kuitenkin onnistuivat luomaan Meta-MCTS avulla aloituskirjan, joka vahvistaa MCTS-menetelmää 7x7-laudalla. MCTS-menetelmä yhdessä Meta-MCTS-menetelmän avulla luodun aloituskirja on paljon vahvempi kuin pelkkä MCTS-menetelmä [Chou et al., 2011].

6. Muut tavat tehdä tietokone-go-ohjelmia

Tähän lukuun on kerätty tapoja tehostaa tai tehdä tietokone-go-ohjelmia. Joitakin näistäkin tavoista voi käyttää yhdessä MCTS-menetelmän kanssa, mutta tekniikat toimivat myös ilman MCTS-menetelmää. Ensimmäiseksi esitellään neuroverkot, jota ei ole paljon viimevuosina edes tutkittu tietokone-gossa. Sitten pohditaan hieman millaisia enemmän ihmisen kaltaisia tekoälytekniikoita on yritetty luoda voittamaan ihmispelaajat. Seuraavaksi esitellään kuviokirjastoja, joita on käytetty go-pelin yhteydessä jo heti ensimmäisistä tietokone-go-ohjelmista lähtien. Viimeisenä esitellään innovatiivista keksintöä, joka tietokannan avulla ennustaa silmänliikkeiden avulla seuraavaa siirtoa go-pelissä.

6.1. Neuroverkot

Ihmispelaajat ovat hyviä tunnistamaan kuvioita, mutta tietokoneet puolestaan ovat juuri kuvioiden tunnistamisessa ja hyvän siirron valitsemisessa huonoja [Richards et al., 1998]. Koska neuroverkot on todettu hyviksi erilaisten kuvioden tunnistamisessa, Richards ja muut [1998] tutkivat, kuinka ne toimisivat go-pelissä. He muokkasivat SANE-ohjelmaa (Symbiotic adaptive neuro-evolution), joka pystyy kehittämään pienillä laudoilla pelaavia verkkoja ilman esiohjelmoitua tietämystä go-pelistä.

Neuroverkot jäljittelevät ihmisen aivoja: neuroneita ja niiden välisiä kytköksiä. Nämä keinotekoiset neuronit sisältävät syötteen ja tulosteen. Tuloste voidaan yhdistää muiden neuronien syötteisiin. Neuroverkot oppivat niille annetuilla esimerkeillä ja ne yrittävät löytää vastaukset, jotka pienentävät virheiden määrää. [Poole et al., 2014]

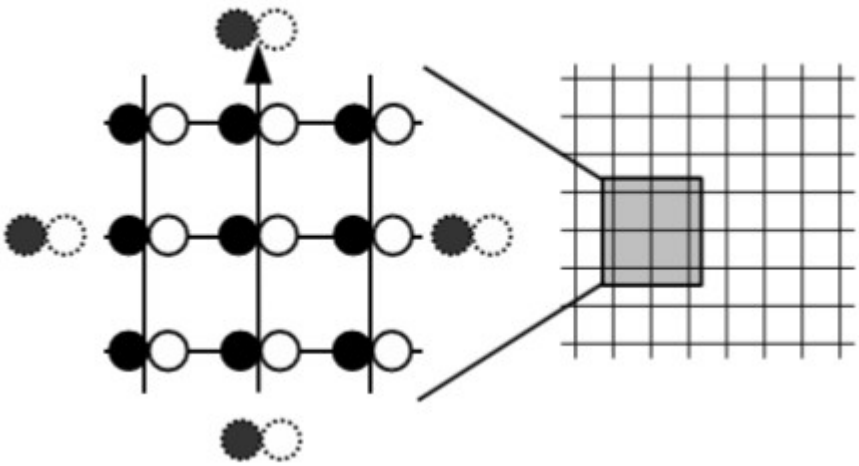
SANE-ohjelma pystyi kehittymään muutamassa sadassa generaatiossa 9x9-laudalla niin, että se pystyi voittamaan yksinkertaisen tietokonevastustajan [Richards et al.,1998]. Richards ja muut [1998] käyttivät vastustajana Wally-ohjelmaa, joka oli hyvä ensivastustajaksi, sillä se ei ole liian hyvä ohjelma, mutta kuitenkin hieman haastava. SANE pystyi nopeasti luomaan strategian, jolla se lopulta voitti Wallyn 75% peleistä. Ongelmana tässä on vain se, että ohjelma loi strategian, joka on hyvä Wallyä vastaan. Ohjelman pitäisi pelata monia eri vastustajia vastaan, jotta se loisi hyvän strategian, eikä vain keskittyisi yhden ohjelman heikkouksiin. Ohjelman kehittäminen on myös vaikeaa, koska se pystyy jo osaamisellaan voittamaan vastustajan ilman lisäkehittymistä. [Richards et al.,1998]

SANE pystyi kehittämään hyviä strategioita ilman etukäteistietämystä pelaamalla go-peliä. Esimerkiksi kehittymisen alussa neuroverkkojen siirrot olivat satunnaisia, kun valittiin ensimmäistä siirtoa. Siirto tapahtui usein reunoille tai niiden läheisyyteen ensimmäisinä generaatioina (koska reunapaikkoja on enemmän kuin keskuspaikkoja 9x9-laudalla), mutta koska neuroverkot huomasivat aloitussiirron tekemisen reunaan johtavan usein ottelun häviämiseen, ne joutuivat muuttamaan strategiaa. Muutaman generaation päästä aloitussiirrot siirtyivät sisemmälle lautaan ja

myohemmin kaikki parhaat neuroverkot aloittivat pelaamisen keskeltä lautaa, joka on hyvä strategia aloittaa. Richardsin ja muiden [1998] mielestä on erityisen hienoa, että SANE kehitti strategian aivan itse päätellen, mitkä siirrot johtivat voittoon ja mitkä häviöön. [Richards et al., 1998]

Richards ja muut [1998] ennustivat, että 13x13-laudalla SANE:n kehittymiseen vaaditaan muutamia tuhansia generaatioita ja 19x19-laudalla jopa kymmeniätuhansia. Lubberts ja Miikkulainen [2001] ovat sitä mieltä, että 19x19-laudalle kehittyvän neuroverkon tekeminen ei ole vielä järkevää, koska generaatioita tarvitaan liikaa. Tulokseksi saatava neuroverkko olisi myös todella iso. Neuroverkoissa on myös se huono puoli, ettei samaa neuroverkkoa pysty käyttämään eri kokoisilla laudoilla, vaan evoluutio ja kehittyminen pitää jokaiselle lautakoolle tehdä erikseen [Stanley and Miikkulainen, 2004].

Stanley ja Miikkulainen [2004] kehittivät harhailevaa silmä -neuroverkkoa (roving eye) go-peliin. He halusivat luoda neuroverkon, joka pystyy pelaamaan go-peliä minkä kokoisella laudalla vain. Harhaileva silmä ei näe koko lautaa kerralla niin kuin neuroverkot yleensä, vaan sillä on pieni näkökenttä, joka pystyy skannaamaan lautaa miten vaan. Neuroverkko kehitettiin NEAT-menetelmän (NeuroEvolution of Agmenting Topologies) avulla. Harhailevalla silmällä oli muisti, jonka avulla se pystyi muistamaan laudan muut osat, ja skannaamalla lautaa se päätti, minne ja milloin siirto tehdään. Kuvassa 16 on harhailevan silmän näkökenttä; Harhaileva silmä näkee 3x3-kokoisen alueen pelilaudalta ja näiden 9 pisteen kohdalla harhailevalla silmällä on yksi musta ja yksi valkoinen sensori. Stanley ja Miikkulainen [2004] peluuttivat menetelmää ensiksi GnuGo:ta vastaan 5x5-laudalla ja näin kehittivät harhailevaa silmää, jonka jälkeen he vaihtoivat laudan kooksi 7x7-laudan. Nyt he testin vuoksi kehittivät kaksi erillistä neuroverkkoa: yksi, joka oli kehitetty 5x5- sekä 7x7-lautojen GnuGo:ta vastaan, ja toinen vain 7x7-laudalle. Stanley ja Miikkulainen [2004] huomasivat, että neuroverkko, joka oli kehittynyt molemmilla laudoilla pärjäsi paremmmin. [Stanley and Miikkulainen, 2004]



Kuva 16: Harhailevan silmän näkökenttä [Stanley and Miikkulainen, 2004].

6.2. Agentit ja oppiminen

Silver ja muut [2012] loivat sovelluskehyksen, joka edustaa laajaa valikoimaa erilaisia algoritmeja, mukana on myös MCTS. Sovelluskehys käyttää väliaikainen ero (temporal difference) -hakua, jossa käytetään arvofunktion lähentämistä (value function approximation), jotta voidaan yleistää yhteen kuuluvia tiloja, sekä bootstrapping-menetelmää, jotta voidaan vähentää arvioitujen arvojen vaihtelua. Sovelluskehys käyttää Monte Carlo -simulaation sijasta väliaikainen ero -oppimista. [Silver et al., 2012]

Väliaikainen ero -haku keskittyy go-pelissä siihen, kuinka suoriudutaan hyvin nyt. Haku tarjoaa kehitystä pelitilanteiden arvioimisen laatuun, eikä arvioi samanlaisia arvoja jokaiselle pelitilanteelle. Haussa luodaan arvofunktio, joka kehittyy pelin edetessä. Näin keskitytään juuri pelattavan pelin taktiikoihin ja strategioihin. Go-peliin on vaikea luoda arvofunktio, samalla lailla kuin MCTS-menetelmään on vaikea luoda simulaatiostrategiaa. Väliaikainen ero -haussa arvioidaan arvot, jotka alkavat pelitilanteesta ja jatketaan pelin loppuun asti. Tämä toteutetaan kouluttamalla arvofunktio uudestaan pelin aikana. Väliaikainen ero -oppiminen tapahtuu pelaamalla itsepelejä, jotka alkavat tämänhetkisestä pelitilanteesta. Itsepelit tapahtuvat kesken pelin kestäen muutamia sekuntteja. [Silver et al., 2012]

Silver ja muut [2012] testasivat väliaikainen ero -hakua Fuego 1.0 -ohjelman UCT-hakua vastaan, jota he kutsuivat vanilja-UCT-hauksi. Vanilja-UCT toimi pelkällä UCT-algoritmilla, josta RAVE ja kaikki muu heuristinen tietämys oli poistettu. Kun laudan koko oli erittäin pieni, 5x5, niin vanilja-UCT-haku pelasi melkein täydellistä peliä, jolloin TD-haku jäi selvästi huonommaksi. Kun laudan kokoa suurennettiin 7x7-laudaksi, niin vanilja-UCT-haku ja TD-haku olivat tasaväkisiä. Mitä suuremmaksi lautaa muutettiin, sitä paremmin TD-haku pärjäsi vanilja-UCT-hakuun verrattuna. Kuitenkin TD-haku oli 3-4 kertaa hitaampi kuin vanilja-UCT-haku. Hyviä puolia Monte Carlo -puuhaussa on se, että se yleensä pelaa simulaatioita paljon enemmän sekunnissa kuin TD-haku, ja jos sillä olisi loputon määrä aikaa ja muistia, niin MCTS löytäisi optimaalisen menettelytavan. [Silver et al., 2012]

Marcolino ja Matsubara [2011] tutkivat, kuinka erilaisten agenttien käyttäminen yhdessä MCTS:n kanssa toimii. He ottivat vertailukohteeksensa Fuego-ohjelman, joka on parhaita go-peliä pelaavia ohjelmia. Marcolino ja Matsubara rakensivat 120 agenttia, jotka toimivat yhdessä Fuegon kanssa ja pelasivat vastustajanaan Fuego ilman agentteja. Näin saatiin vertailtua, onko Fuego agenttien kanssa parempi kuin pelkkä Fuego.

Marcolino ja Matsubara [2011] esittävät, että paras ryhmä ei välttämättä koostu parhaista yksilöistä, vaan siitä, että ryhmäläisten taidot ovat erilaisia. Tätä he yrittivät saavuttaa myös omilla go-peliä pelaavilla agenteilla. Idea on innovatiivinen ja vaikuttaa yllättävän toimivalta. Agentit olivat erityyppisiä go-pelin pelaajia. Kaikkien 120 agentin käyttö yhdessä ei tuottanut parasta tulosta. Agentteja valittiin agenttitietokannasta satunnaisesti ja Marcolino ja Matsubara loivat eri

menetelmillä ryhmiä agenteista ja testasivat niiden voittoprosentin Fuegoa vastaan. Mielenkiintoinen huomio oli se, että yksittäinen agentti, joka yksin pärjäsi huonosti Fuegoa vastaan, saattoi silti nostaa ryhmän voittoprosenttia, kun agentti lisättiin pelaavaan ryhmään [Marcolino and Matsubara, 2011].

Marcolino ja Matsubara [2011] väittävät, että monien agenttien käyttö go-peliä pelaavissa ohjelmissa on uusi läpimurto. He ovat varmoja siitä, että kyseinen menetelmä tulee voittamaan Fuegon. Kova väite, mutta he eivät ole vielä testanneet menetelmää muuten kuin Fuegoa vastaan. Ihmispelaajien kanssa tilanne voi olla aivan toinen. Vaikka Fuego voitetaan, ei se tarkoita, että ohjelma voittaisi ihmispelaajat, jotka Fuego on voittanut. Marcolino ja Matsubara [2011] eivät vielä pystyneet edes sanomaan, että onko ryhmä oikein valittuja agentteja parempi verrattuna yhteen parhaaseen agenttiin, koska erot olivat liian pienet.

6.3. Kuviokirjastot

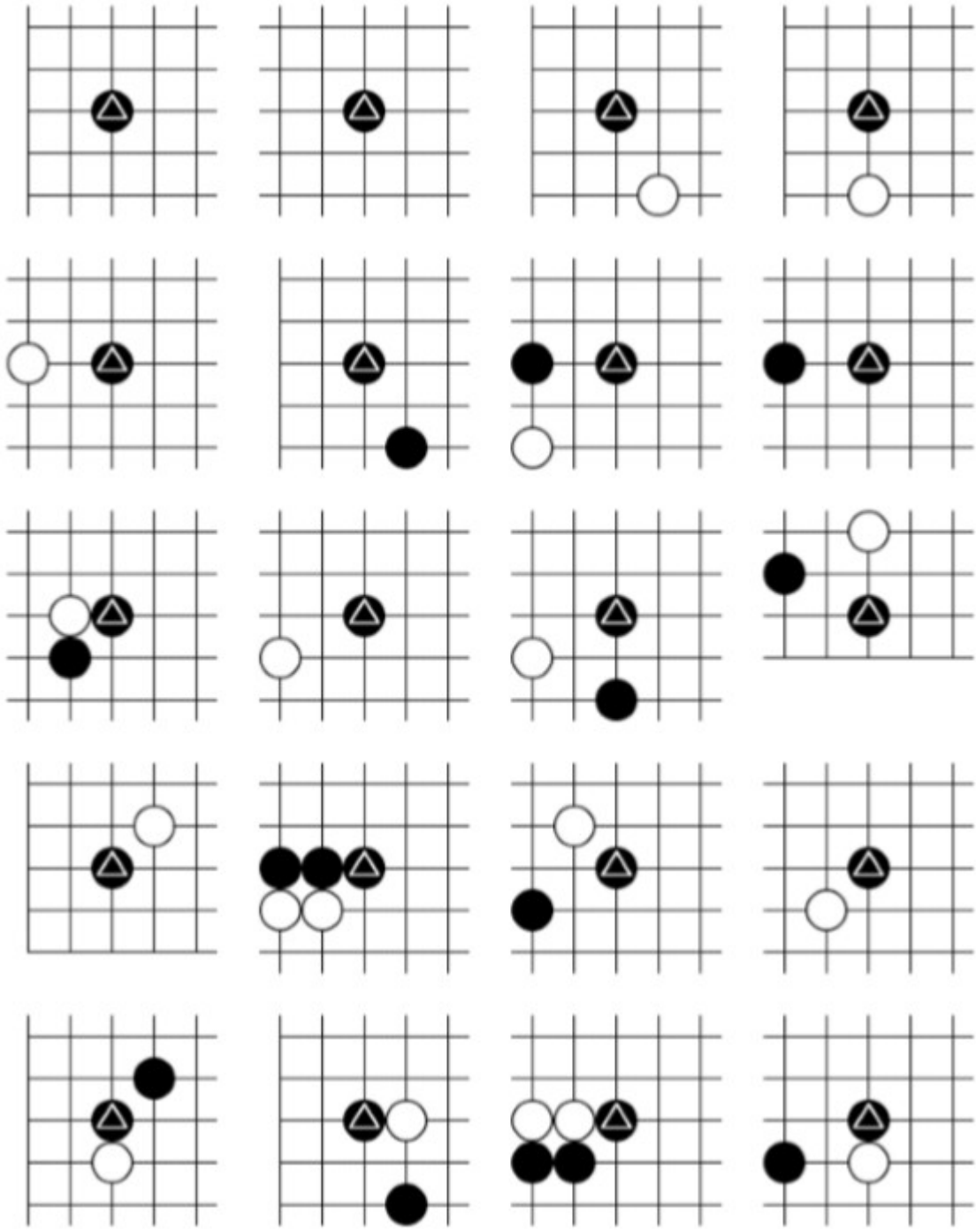
Kuviokirjastojen (pattern library) käyttö tietokone-gossa on yleistä, koska go-pelissä on paljon mahdollisia pelikuvioita. Jo ensimmäinen tietokone-go-ohjelma vuodelta 1968 käytti kuviontunnistusta hyväkseen [Sensei's Library, 2014]. Erilaista kuviotietämystä on käytetty tietokone-gossa siitä pitäen. Chenin ja muiden [2008] mielestä kuviotietämys on tärkeää sekä MCTS-ohjelmilla ja muissa ohjelmissa, jotta saataisiin tasokkain mahdollinen tietokone-go-ohjelma luotua. Esimerkiksi GNU GO -ohjelma sisältää kuviotietokannan siirtojen valitsemiseen [Wang et al., 2011]. Myös MoGo- ja Fuego-ohjelmat käyttävät kuvioita parantamaan satunnaisten pelien simulaatioita [Wang et al., 2011].

Kuviokirjastojen ja tietokantojen ongelmana on kuitenkin se, että kuvioita on go-pelissa paljon, eikä ole sellaista yleistä kuviota, joka esiintyisi ihan jokaisessa pelissä. Siksi onkin tärkeää, kuinka kuviot tunnistetaan ja kuinka niitä hyödynnetään. Kuvioita käyttävät sekä ihmispelaajat että tietokone-go-ohjelmat [Liu et al., 2011]. Ihmispelaajille kuvioiden tunnistaminen on helppoa. Koska kivet eivät liiku pelilaudalla, pystyvät ihmispelaajat helposti ennakoimaan laudan tilanteita eteenpäin. Aloittelevat pelaajatkin pystyvät ennakoimaan 15-20 siirtoa eteenpäin yhdestä tilanteesta ja jopa 50 siirtoa tai enemmän ns. tikapuu-tilanteessa. Ihmiset pystyvät erottamaan pelilaudalta monimutkaisia riippuvaisuuksia kivien väliltä. Tämä kivien rippuvaisuuksien havannointi on vaikeaa mallintaa tietokoneelle. [Müller, 2002] Hyvä kuviotietämys on vaikeaa kuvata määrällisesti ja ohjelmoida. Kuvioiden väliset vuorovaikutukset voivat johtaa myös ennalta arvaamattomaan käytökseen. [Silver et al., 2012]

Ohjelmien kuviotietämys saadaan joko jo valmiista kuviokirjastoista, koneoppimalla ne, ihmisasiantuntijuudella, vahvistusoppimisella (reinforcement learning) tai hakemalla kuviot pelitallenteista tilastollisesti [Liu et al., 2008]. Kuvassa 17 on 20 yleisintä go-pelin kuviota ammattilaispeleistä, jotka Liu ja muut [2008] löysivät. Vaikka hakee kuviot kuinka suuresta pelitietokannasta tahansa, niin useimmat kuviot eivät tapahdu peleissä usein ja vain pieni määrä

kuvioista ilmestyy useissa peleissä [Liu et al., 2008]. Tiedonhaussa on myös kaksi suurta ongelmaa: kuvion käyttötapaa ei välttämättä pystytä ymmärtämään ellei kuviolla ole suhteellisen suurta määrää esiintymisiä; toiseksi on mahdotonta tutkia kaikkia kuvioita, joita go-pelissä on, koska jokaisessa uudessa pelissä on yleensä kokonaan uusi ennennäkemätön kuvio [Liu et al., 2008]. Hyvissä tietokone-go-ohjelmissa voi olla joseki-tietokantoja, joihin on tallennettu 5000 – 50000 siirtoa [Richards et al., 1998].

Wang ja muut [2011] testasivat 4x4-kuvioita tavallisen 3x3-kuvion sijasta. Vaikeutena tässä oli se, että 4x4 ei ole symmetrinen, ja se on siksi hankalampi toteuttaa. Koska 4x4 on isompi kuin 3x3, niin sen etuna on se, että se sisältää enemmän tietoa [Wang et al., 2011]. Mitä suurempi sen parempi periaatteessa, mutta jo 5x5-kuviot vievät liikaa muistia. Se sisältäisi symmetrian, mutta suuruuden takia kuvioita on enemmän ja prosessointi vie liian kauan aikaa. Liian suuri kuvio saattaa myös kärsiä poikkeuksellisesta reunatietämyksestä, joka johtaa vääriin johtopäätöksiin. 4x4-kuviointi pystyy parantamaan MCTS-algoritmia hiukan ja varsinkin isoilla laudoilla se toimii paremmin kuin 3x3-kuviointi. [Wang et al., 2011]



Kuva 17: 20 yleisintä go-pelin kuviota ammattilaispeleistä [Liu et al., 2008]

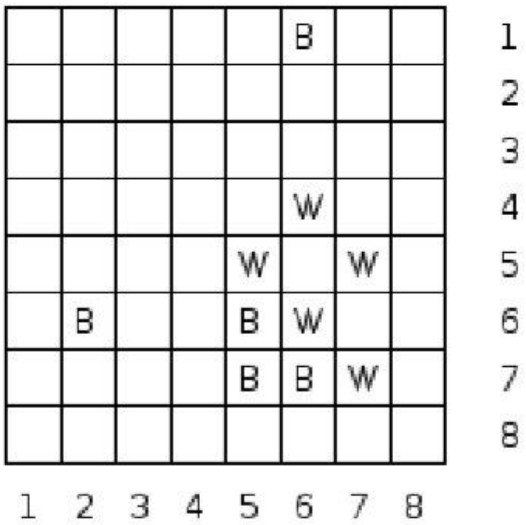
6.4. Siirtojen ennustaminen silmän liikkeiden avulla

Bossomaier ja muut [2012] tutkivat, kuinka go-pelissä pystyy silmänliikkeiden avulla ennustamaan, mihin seuraava siirto todennäköisesti tehdään. He käyttivät kahta mallia, CHUMP ja CHREST. CHREST tunnistaa lohkot (chunks) ja CHUMP käyttää näitä lohkoja ennustamaan mahdollisen seuraavan siirron [Bossomaier et al., 2012]. Bossomaierin ja muiden [2012] tavoite oli tutkia, kuinka hyvin lohkot pystyvät ennustamaan matalan tason toteutuksella, minne seuraava siirto tehdään paikallisella alueella.

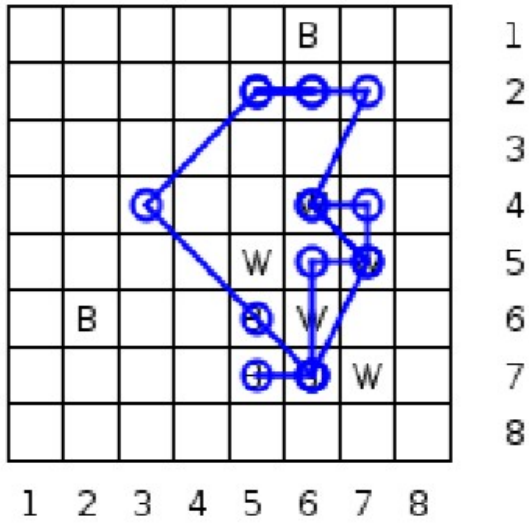
Bossomaier ja muut [2012] tutkivat ensin, kuinka ihmispelaajien katseet käyttäytyivät go-peliä pelatessa. He totesivat, että huippupelaajat tunnistavat alueen, johon siirto kannattaa tehdä alle sekunnissa. Yleensä tämä alue on ensimmäinen, johon heidän huomionsa kiinnittyy. On huomattu, että pelaajien huomio kiinnittyy kivien välisiin alueisiin, eikä itse kiviin. Pelaajat katsovat vain tietyn osan laudasta ennen kuin tekevät siirtonsa ja sen jälkeen katsovat kokonaistilanteen laudalla. [Bossomaier et al., 2012]

CHREST on itseohjautuva ja dynaaminen järjestelmä. CHREST koostuu neljästä osasta, joiden avulla se tunnistaa lohkoja. CHREST tallentaa lohkot syrjintäverkosta (discrimination net), jonne niitä lisätään oppimisen jälkeen. Syrjintäverkko on puumainen tietorakenne, jonka solmuja ovat lohkot. Lohkot opitaan tietokannoista, joihin on tallennettu tietoa go-pelistä. [Bossomaier et al., 2012]

CHUMP-mallin avulla Bossomaier ja muut [2012] saivat ennustusprosentikseen 50%. Eli ohjelma osasi ennustaa oikean siirron puolessa testitapauksista. Ohjelma kuitenkin tutki vain tilanteita, eikä varsinaisesti pelannut go-peliä. Kuvissa 18 ja 19 on esimerkit, millaisia tilanteita Bossomaierin ja muiden ohjelma tutki. Bossomaierin ja muiden [2012] ohjelma onkin hyvin alkutekijöissään, ja sen hyötyä laajemmin ei voi vielä tietää varmasti. Heidän mukaansa tämä on kuitenkin tärkeää tutkimusta, jonka avulla tekoäly voisi toimia samalla lailla kuin ihmisasiantuntijuus.



Kuva 18: Esimerkki go-pelipaikoista pienennetyllä 8x8-laudalla [Bossomaier et al., 2012] (Huomioi, että poikkeuksellisesti nappulat ovat ruuduissa, eikä viivojen leikkauskohdissa).



Kuva 19: Esimerkki katseen kiinnittymisestä laudalla [Bossomaier et al., 2012].

7. Päätelmät

Tietokone-gon tavoitteena on ollut jo muutaman vuosikymmenen ajan ollut ohjelma, joka pystyy pelaamaan go-peliä huippuammattilaisen tasolla 19x19-laudalla. Välitavoitteina on ollut eri kokoisten lautojen ratkaisu sekä eri tasoisten pelaajien voittaminen. Tällä hetkellä 6x6-laudalle pystytään luomaan täydellinen siirto jokaiseen pelitilanteeseen. Tietokone-gon taso on vasta harrastelijadanien tasoilla. Vielä kestää varmasti vuosikymmen ennen kuin tietokoneet ovat huippuammattilaisten tasolla, ellei jotain uuttaa pätevää menetelmää keksitä.

Tietokone-gossa tämän hetken vahvin menetelmä on kiistämättä MCTS-algoritmi. Kuitenkin jotkut tutkijat ovat sitä mieltä, että MCTS-algoritmin hyödyt on jo käytetty. Cook [2010] on sitä mieltä, että MCTS:n kehitys on hidastunut. Voi olla, että varsinaisen MCTS-menetelmään ei ole keksitty mitään uutta, mutta parannuksia on saatu lisäilemällä siihen erilaisia algoritmeja ja menetelmiä. Vaikuttaa myös siltä, että nyt ollaan vasta ottamassa käyttöön kaikki MCTS-algoritmin hyödyt, koska vasta viime vuosina on pystytty MCTS-algoritmin avulla voittamaan ammattilaispelaajia. Kroeker [2011] uskoo, että Monte Carlo -puuhaulla saadaan voitto tietokoneille go-pelissä. Hän kertoo, kuinka muutamassa vuodessa tietokoneet ovat editystyneet huimasti Monte Carlo -puuhaun myötä. Edistys on ollut kohtuullista verrattuna esimerkiksi edellisen kymmenen vuoden edistymiseen, mutta nyt taas aletaan olla siinä vaiheessa, että jotain uutta pitäisi keksiä, jotta kehitys pysyisi nopeana.

MCTS-algoritmin skaalautuvuus tarkoittaa sen kykyä pelata paremmin silloin, kun lisätään tietokoneiden tehoa tai laskenta-aikaa. Bourkin ja muiden [2010] mukaan skaalautuvuus esitetään usein MCTS-algoritmin etuna. Skaalautuvuus näkyy käytännössä MCTS-algoritmissa, kun käytetään rinnakkaislaskentaa. Koska tiedetään, että rinnakkaistaminen on melko tehokasta, Bourki ja muut [2010] huomauttavat, että kun rinnakkaistamisen tehokkuus ja MCTS-algoritmin skaalautuminen otetaan huomioon, niin ohjelmien tulisi olla jo paljon vahvempia kuin ihmispelaajat. Bourki ja muut [2010] testasivat, että MCTS-algoritmin skaalautuvuus selvästi pienenee, kun tietokoneiden määrä kasvaa. Segalin [2010] mielestä tämä näkyy selvästi, kun katsoo vuoden 2009 tietokoneolympialaisten tuloksia 19x19-laudalla pelattavan go-pelin osalta. Zen-ohjelma tuli ensimmäiseksi käyttäen yhtä neljän ytimen systeemiä. Toiseksi tuli Fuego-ohjelma, joka käytti kymmentä 8 ytimen systeemiä. Kolmanneksi jäi MoGo, jolla oli käytössään kaksikymmentä 32 ytimen systeemiä. Eli ytimien määrä ei taannut voittoa. Tärkein asia tietokone-gon rinnakkaistamisessa on se, kuinka käyttää lisätehot, eikä vain tee samaa asiaa monta kertaa uudelleen. [Segal, 2010]

Skaalautuvuus MCTS-ohjelmissa sisältää joitakin etuja, mutta sillä on selviä rajoituksia. Vaikka käytettäisiin suurta muistia ja suurta määrä tietokoneita, niin jotkin tehtävät ovat sellaisia, joita MCTS-ohjelma ei osaa ratkaista [Bourki et al., 2010]. Jotkin tutkijat ovat eri mieltä ja

esimerkiksi Silver ja muut [2012] väittävät, että MCTS pystyy loputtomilla resurseilla löytämään optimaalisen ratkaisun. Tietenkään loputtomat resurssit eivät tule tulevaisuudessakaan olemaan mahdollista toteuttaa, joten sillä ei ole periaatteessa merkitystä MCTS-menetelmän hyvyyden kannalta. Sellaisen tekniikan kehittäminen, jota pystyittäisiin rinnakkaistamaan helposti, sekä mahdollisimman monelle tietokoneelle niin, että pelin taso vain paranisi, toimisi hyvin go-pelissä, mutta tällaisen tekniikan kehittäminen on helpommin sanottu kuin tehty.

MCTS-menetelmässä on puutteita ja ansioita, mutta tutkijat ovat usein eri mieltä molemmista. Se kuitenkin hyväksytään kiistatta parhaaksi menetelmäksi tietokone-gossa tällä hetkellä. MCTS-menetelmän käytetyimmästä versiosta UCT-algoritmista ei myöskään päästä yhteisymmärrykseen. Se on vaikea rinnakkaistaa ja joistakin se on liian hidas. UCT-algoritmia pidetään kuitenkin parhaana versiona MCTS-menetelmää. RAVE on luotu parantamaan UCT-algoritmia, mutta joskus RAVE heikentää ohjelmien tasoa huonommaksi kuin mitä pelkällä UCT-algoritmilla saataisiin aikaiseksi. Tämän takia pitäisi joko keksiä, milloin RAVE:a ei pelissä käytetä, tai sellainen lisä, joka ei huononna go-peliohjelmien tasoa. Olisi tärkeää tutkia tarkoin, minkälaisissa tilanteissa RAVE tekee väärän johtopäätöksen ja milloin se toimii erityisen hyvin, sekä osata opettaa ohjelmalle erottamaan tilanteet, joissa RAVE:a on soveliasta käyttää.

Lisäykset MCTS-menetelmään kuten solmulaajennus ja Meta-MCTS vaikuttavat mielenkiintoisilta kokeiluilta, mutta eivät todennäköisesti tuo mitään mullistuksia tietokone-gohon. Solmulaajennus vaikuttaa turhan monimutkaiselta ja vaikka Yajima ja muut [2011] ovat sitä mieltä, että solmulaajennus on kannattava lisä UCT-algoritmiin, niin itse olen eri mieltä. Kaikkea ei milläään kannatta lisätä yhteen ja toivoa, että menetelmä paranisi. Voi olla, että jos käyttää pelkkää UCT-algoritmia, solmulaajennuksen lisääminen voi kannattaa, mutta en usko, että yhdessä kaikkien muiden tekniikoiden kanssa se parantaa ohjelmaa. Meta-MCTS -menetelmä suurin ongelma on se, että se on hidas ja menettelytapa muuttuu liian monimutkaiseksi. Meta-MCTS oli jo liian hidas pienellä laudalla ja 19x19-laudalla sen tehokkuus ja nopeus laskisi huimasti, eikä sen käyttö olisi mielekästä.

Bossomaierin ja muiden [2012] mielestä tietokoneiden kyky hankkia ja hyödyntää ihmisasiantuntemusta on puuttellinen. He jatkavat, että esimerkiksi shakissa tietokone toimii haulla, kun taas ihmiset hyödyntävät kuviomuistia ja heidän kokemustaan samanlaisista aikaisemmin kohdatuista tilanteista. Bossomaierin ja muiden [2012] mielestä go-peli on mielenkiintoinen, koska se on hyvä alue tutkia ihmiskognitiota, sekä on olemassa mahdollisuus, että tietokone päihittää ihmispelaajat käyttämällä ihmisten strategiaa. Tietokone, joka osaa käyttää ihmisten strategiaa, on mielenkiintoinen ajatus, mutta kuulostaa vaikealta toteuttaa. Kannatan mieluummin sitä, että yritetään rakentaa tietokone, jonka strategia ei ole samanlaista kuin ihmisillä, vaan yritetään parantaa tekoälystrategioita. Tällöin voidaan päästä ratkaisuihin, joihin ihmisaivoilla ei kyetä pääsemään.

Bossomaier ja muut [2012] pohtivat, että havaintokyky ja tieto tulee olla parempi go-pelissä kuin shakissa, koska hakupuu on myös paljon suurempi go-pelissä. Go-peliä usein verrataan shakkiin, vaikka peleillä ei kovin paljon ole yhteistä. Tämä voi johtua tietysti siitä, että molemmat on kahden hengen pelejä ja niissä on molemmissa valitaan mustat tai valkoiset nappulat. Molempia pelataan myös laudoilla, mutta laudat ovat hyvin erilaiset. Siksi olisikin mielenkiintoista tietää, onko ammattilainen shakinpelaaja myös hyvä go-pelissä (tai toisinpäin) Olisi myös mielenkiintoista tietää, saako shakkia pelaavista tietokoneista vielä parempia MCTS-menetelmää käyttämällä. Voi olla, että MCTS-algoritmin lisääminen turhaa, koska shakin ammattilaispelaajat on pystytty voittamaan jo vuosikymmeniä sitten. Tietenkin voi olla, että shakki saadaan ratkaistua samalla menetelmällä, jolla päästään tietokone-gossa lähes huippuammattilaistasolle.

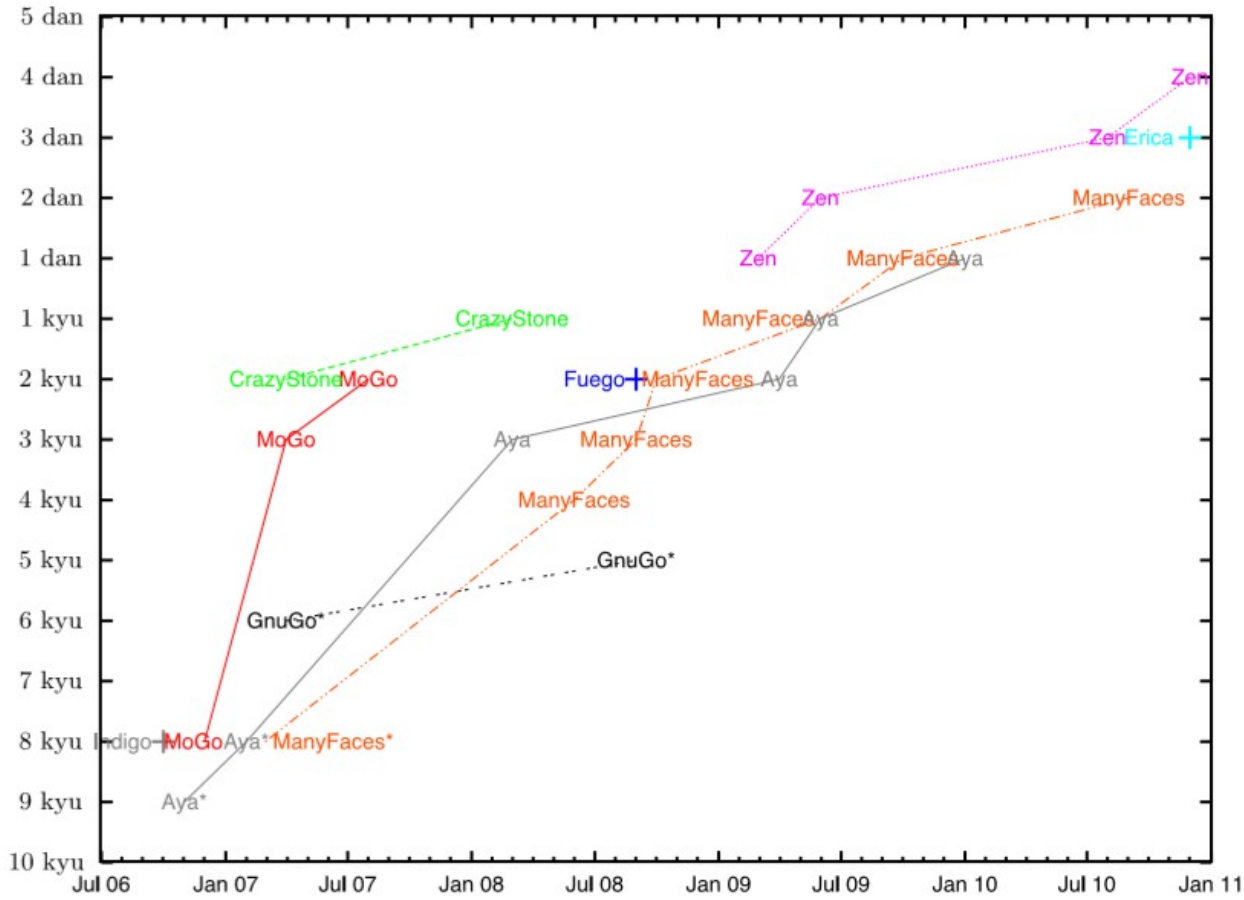
Monet tutkijat, esimerkiksi Bossomaier ja muut [2012] ja Marcolino ja Matsubara [2011], yrittävät luoda go-peliä pelaavista tekoälyohjelmista enemmän ihmisen kaltaisia. Heidän mielestään ohjelmien osaamistason tulee vaihdella jopa kesken pelinkin, jotta he muistuttaisivat enemmän ihmisten kaltaisia pelaajia. Marcolinon ja Matsubaran [2011] mukaan pelkät ”vahvat” siirrot, joilla valloitetaan vastustajan kivi tai kiviryhmiä, eivät takaa go-pelissä voittoa, vaan tekevät pelin huonoksi ja epäluonnolliseksi. Tämä väite kertoo siitä, ettei go-pelille ole olemassa minkäänlaista vedenpitävää voittostrategiaa, joka toimisi joka pelissä. Vaikuttaa siltä, ettei jollain pelityylilläkään tule varmaa voittoa. Kaikki pelissä riippuu siitä, ketkä pelaavat vastakkain.

On harmillista, ettei uudempia neuroverkko kokeiluja go-pelin suhteen ole tehty. Joko on tultu sellaiseen tulokseen, etteivät neuroverkot toimi go-pelin yhteydessä, tai ne vievät liikaa resursseja. Nyt olisi kuitenkin mahdollista peluuttaa neuroverkkoja MCTS-metodia käyttäviä ohjelmia vastaan, jolloin ongelmana ollut vain vastustajan strategian heikkouksien oppiminen ei tapahtuisi ja neuroverkoille olisi mahdollista saada parempia harjoitusvastustajia kuin mitä 2000-luvun alussa oli. Koska tavallinen neuroverkko pitää kouluttaa pelaamaan kaikkia lautakokoja erikseen, olisi melkein pakko aloittaa suoraan 19x19-laudalla, jolloin tälläisen neuroverkon tekeminen voi olla liian suuri työtaakka aloittaa tyhjästä.

Kuviokirjastoja tai tietokantoja on käytetty jo tietokone-gon alkumetreiltä asti. Varsinkin fuseki-tietokanta vaikuttaa järkevältä lisältä, koska aloitussiirtojen hyötyjä on vaikeaa laskea tietokoneilla. Kuviokirjastojen käyttö kuitenkin tuottaa myös paljon ongelmia. Kuvioita on liikaa go-pelissä, jotta pelkällä kuviokirjastolla pystyttäisiin luomaan toimiva ohjelma. Ihmispelaajat ovat erinomaisia soveltamaan omaa tietämystään vanhoista kuvioista uusiin pelissä ennaltanäkemättömiin kuvioihin, mutta tietokoneilta tämä kyky puuttuu. Kuviotietämys lisänä jonkun muun tekniikan kanssa on usein järkevää, mutta lopullisena tavoitteena voisi pitää ohjelmaa, joka pystyisi ilman kuviotietämystä päättelemään hyviä siirtoja kaikissa tilanteissa, koska kuviokirjastolla ei tähän yksinkertaisesti pystytä.

Ohjelma, joka voittaa aina maailman parhaimman pelaajan, on go-pelin tapauksessa todennäköisesti mahdoton toteuttaa. Tämän päivän yritykset 19x19-laudalla ovat vielä kaukana edes yhdestä voitosta, jos pelataan ilman tasoitusta. Mielestäni monet ovat liian optimistisella kannalla siitä, että go-pelin ratkaisemiseen löytyy joku selvä ratkaisu. Fuego-ohjelman yksi tekijä Muller on sitä mieltä, että ratkaisu löytyy silloin, kun MCTS-menetelmään lisätään ja yhdistetään joitain muita menetelmiä [Kroeker, 2011]. Silverin ja muiden [2012] sovelluskehys, joka sisälsi kymmeniä eri algoritmeja go-pelin pelaamiseen on yksi mahdollinen tietokone-gon tulevaisuus. Useiden eri menetelmien käyttö on mahdollista, mutta hankaluudeksi varmasti nousee toteutustavan monimutkaisuus. Go-peliä pelaavat ohjelmat muuttuvat liian hitaiksi ja virheiden määrä kasvaa sitä mukaa, mitä monimutkaisemmaksi ohjelma tai algoritmi muuttuu.

Kuvassa 20 on koottu miten tietokone-go-ohjelmien taso on noussut vuosina 2006 – 2011. Kuva on piirretty turhan optimistisen näköiseksi, koska kyu-tasolla yksi nousu ei ole sama asia kuin dan-tasojen välit. Kuvasta kuitenkin näkee, kuinka suuri kehitys tapahtui vuonna 2006, kun esimerkiksi MoGo nousi viisi tasoa korkeammalle vuodessa. Kehitys on hieman hidastunut, mutta vielä ei kuitenkaan olla lähelläkään ammattilaisdan-tasoa.



Kuva 20: Tietokone-go-ohjelmien tason kehitys [Gelly and Silver, 2011].

8. Yhteenveto

Tekoäly, joka pystyy pelaamaan samalla tasolla huippuammattilaisten kanssa, on vielä keksimättä. Go-peli on hyvä tapa parantaa vanhoja tekoälytekniikoita sekä kehitellä kokonaan uusia tekoälytekniikoita, joita voi käyttää muillakin aloilla. Kun ammattilaispelaajien tasolle päästään, niin on yksi suuri tavoite saavutettu tekoälyn osalta.

Käsittelin tutkielmassani erilaisia tekoälytekniikoita, joita go-pelin pelaamisen on kokeiltu. Monte Carlo -puuhaku on suurennuslasin alla tällä hetkellä tietokone-gon parantamisessa, joten tutkielmakin käsittelee pääasiassa sitä ja siihen lisättyjä tekniikoita. Mukana on myös muutama tekniikka, joita voi käyttää yhdessä MCTS-menetelmän kanssa tai sitten yhdessä jonkin muun tekniikan kanssa.

Tekoäly, joka pystyy pelaamaan go-peliä ammattilaistasolla, pitää olla jokin innovatiivinen ratkaisu. Uskon, että ratkaisu löytyy jostain vanhasta tekoälytekniikasta, johon on keksitty jokin uusi innovatiivinen lisäys. Tulevaisuus go-pelin päihittäville tekoälytekniikoille näyttäisi olevan se, että kokeillaan vähän kaikkea MCTS-menetelmän kanssa yhdessä, ja testataan, toimiiko se paremmin kuin pelkkä MCTS-menetelmä. Voi myös olla, että MCTS-menetelmää ei pysty toteuttamaan toimivammin go-pelille ja täytyy keksiä jokin muu ratkaisu. MCTS-menetelmällä varmaan pystyy pelaamaan vielä paremmin ainakin 9x9-lautaan asti.

Go-pelille ei varmaankaan koskaan tule löytymään ilmiselvää voittostrategiaa tai algoritmia, jonka pystyisi vain haun avulla laskemaan parhaat siirrot. On hienoa, että on olemassa tällainen peli, jonka ratkaisua ei löydy helposti, koska tutkijat joutuvat miettimään innovatiivisesti ja varmasti go-pelin takia uusia tekoälytekniikoita keksitään, joita voidaan hyödyntää jollain muulla alueella, vaikka ne eivät go-pelin kanssa toimisikaan.

Viiteluettelo

[Allis, 1994] Louis Victor Allis, Searching for Solutions in Games and Artificial Intelligence. Thesis, 1994. Available as http://fragrieu.free.fr/SearchingForSolutions.pdf. Checked 07.07.2014.

[Bossomaier et al., 2012] Terry Bossomaier, Jason Traish, Fernand Gobet and Peter C. R. Lane, Neuro-cognitive model of move location in the game of go. In: *Proc. of the 2012 International Joint Conference on Neural Networks*, (June 2012), 1-7.

[Bourki et al., 2010] Amine Bourki, Guillame Chaslot Matthieu Coulm, Vincent Danjean, Hassen Doghmen, Jean-Baptiste Hoock, Thomas Hérault,Arpad Rimmel, Fabien Teytaud, Olivier Teytaud, Paul Vayssière and Ziqin Yu, Scalability and parallelization of Monte-Carlo tree search. In: *Proc. of the 7th International Conference on Computer and Games, Revised Selected Papers,* (2010), Springer,48-58.

[Bouzy, 2005] Bruno Bouzy, Move-pruning techniques for Monte-Carlo go. In: *Proc. of the 11th International Conference on Advances in Computer Games* (2005), Springer, 104-119.

[Cazenava, 2009] Tristan Cazenava, Nested Monte-Carlo Search. In: *Proc. of the 21th International Joint Conference on Artificial Intelligence* (2009), 456-461.

[Cazenava and Jouandeu, 2008] Tristan Cazenava and Nicolas Jouandeau, A parallel Monte-Carlo tree search algorithm. In: *Proc. of the 6th International Conference on Computer and Games* (2008), Springer, 72-80.

[Chaslot, 2010] Guillaume Maurice Jean-Bernard Chaslot, Monte-Carlo Tree Search. Thesis, 2010. Available as https://project.dke.maastrichtuniversity.nl/games/files/phd/Chaslot_thesis.pdf. Checked 27.06.2014.

[Chaslot et al., 2008] Guillaume M.J-B. Chaslot, Mark H.M. Winands and H. Jaap van den Herik, Parallel Monte-Carlo tree search. In: *Proc. of the 6th International Conference on Computer and Games* (2008), Springer, 60-71.

[Chen et al., 2008] Keh-Hsun Chen, Dawei Du and Peigang Zhang, A fast indexing method for Monte-Carlo go. In: *Proc. of the 6th International Conference on Computer and Games* (2008), Springer, 92-101.

[Chou et al., 2011] Cheng-Wei Chou, Ping-Chiang Chou, Hassen Doghmen, Chang-Shing Lee, Tsan-Cheng Su. Fabien Teytaud, Olivier Teytaud, hui-Ming Wang, Mei-Hui Wang, Li-Wen Wu and Shi-Jim- Yen, Towards a solution of 7x7 go with meta-MCTS. In: *Proc. of the 13th International Conference on Advances in Computer Games, Revised Selected Papers,* (2011), Springer, 84-95.

[Computer Go, 2014] Human-Computer Go Challenges. Available as http://www.computer-go.info/h-c/ . Checked 21.04.2014

[Cook, 2010] Darren Cook, A human-computer team experiment for 9x9 go. In: *Proc. of the 7th International Conference on Computer and Games, Revised Selected Papers,* (2010), Springer, 145-155.

[Doyle, 2014] Search Methods by Patrick Doyle. Available as http://www.cs.duke.edu/brd/Teaching/Previous/AI/Lectures/Summaries/search.html. Checked 04.06.2014

[Gelly and Silver, 2011] Sylvain Gelly and David Silver, Monte-Carlo tree search and rapid action value estimation in computer go. *Artificial Intelligence* **175,** 11 (July 2011), 1856-1875.

[Gelly et al., 2012] Sylvain Gelly, Levente Kocsis, David Silver and Csaba Szepesvári, The grand challenge of computer go: Monte Carlo tree search and extensions. *Comm. ACM* **53**, 3 (March 2012), 106-113.

[Harré et al., 2011] M.S. Harré, T. Bossomaier, A. Gillet and A. Snyder. The aggregate complexity of decisions in the game of go. *The European Physical Journal B* **80,** 4 (April 2011), 555-563.

[Hashimoto et al., 2011] Junichi Hashimoto, Akihiro Kishimoto, Kazuki Yoshizoe and Kokolo Ikeda, Accelerated UCT and its application to two-player games. In: *Proc. of the 13th International Conference on Advances in Computer Games, Revised Selected Papers,* (2011), Springer,1-12.

[Kroeker, 2011] Kirk L. Kroeker, A new benchmark for artificial intelligence. *Comm. ACM* **54**, 8 (August 2011), 13-15.

[Liu et al., 2008] Zhiqing Liu, Qing Dou and Benjie Lu, Frequency distribution of contextual patterns in the game of go. In: *Proc. of the 6th International Conference on Computer and Games* (2008), Springer, 125-134.

[Lubberts and Miikkulainen, 2001] Alex Lubberts and Risto Miikkulainen, Co-evolving a go-playing neural network. In: *Coevolution: Turning Adaptive Algorithms upon themselves, Birds-of-a-Feather Workshop, Genetic and Evolutionary Computation Conference (2001),*

[Marcolino and Matsubara, 2011] Leandro Soriano Marcolino and Hitoshi Matsubara, Multi-agent Monte Carlo go. In: *Proc. of the 10th International Conference on Autonomous Agents and Multiagent Systems - Volume 1*, 21-28.

[Müller, 2000] Martin Müller, Review: computer go 1984-2000. In: *Proc. of the Second International Conference on Computer and Games, Revised Papers* (2000), Springer, 405-413.

[Müller, 2002] Martin Müller, Computer go. *Artificial Intelligence* **134** (2002), 145-179.

[MCTS.ai, 2014] Monte Carlo Tree Search. Available as http://mcts.ai/index.html .Checked 26.06.2014.

[Niekerk et al., 2012] Francois van Niekerk, Gert-Jan van Rooyen, Steve Kroon and Cornelia P. Inggs, Monte-Carlo tree search parallelisation for computer go. In: *Proc. of the South African Institute for Computer Scientists and Information Technologists Conference 2012*, 129-138.

[Poole et al., 2014] Computational Intelligence, A Logical Approach. Poole, Mackworth and Goebel. Available as http://www.cs.ubc.ca/~poole/ci/lectures/lectures.html. Checked 05.06.2014.

[Richards et al., 1998] Norman Richards, David E. Moriarty and Risto Miikkulainen, Evolving neural networks to play go. *Applied Intelligence* **8**, 1 (January 1998), 85-96.

[Rimmel et al., 2010] Arpad Rimmel, Fabien Teytaud and Olivier Teytaud, Biasing monte-carlo simulations through RAVE values. In: *Proc. of the 7th International Conference on Computer and Games, Revised Selected Papers,* (2010), Springer, 59-68.

[Russel and Norwig, 2009] Stuart Russel and Peter Norvig, *Artificial Intelligence: a Modern Approach (Third Edition)*. Prentice Hall, 2009.

[Segal, 2010] Richard B. Segal, On the scalability of parallel UCT. In: *Proc. of the 7th International Conference on Computer and Games, Revised Selected Papers,* (2010), Springer, 36-47.

[Sensei's Library, 2014] Sensei's Library, GNU Go. Available as http://senseis.xmp.net/. Checked 04.03.2014

[Silver et al., 2012] David Silver, Richard S. Sutton and Martin Müller, Temporal-difference in computer go. *Machine Learning* **87**, 2, 183-219.

[Stanley and Miikkulainen, 2004] Kenneth O. Stanley and Risto Miikkulainen, Evolving a roving eye for go. In: *Proceedings of the Conference on Genetic and Evolutionary Computation, Part II (June 2004), 1226-1238.*

[Tom and Müller, 2010] David Tom and Martin Müller, Computational experiments with RAVE heuristic. In: *Proc. of the 7th International Conference on Computer and Games, Revised Selected Papers,* (2010), Springer, 69-80.

[Tromp and Farnebäck, 2006] John Tromp and Gunnar Farnebäck, Combinatorics of go. In: *Proc. of the 5th International Conference on Computer and Games, Revised Papers* (2006), Springer, 84-99.

[Van Eyck and Müller, 2011] Gabriel Van Eyck and Martin Müller, Revisiting move groups in monte-carlo tree search. In: *Proc. of the 13th International Conference on Advances in Computer Games, Revised Selected Papers,* (2011), Springer, 13-23.

[Wang et al.,2011] Jiao Wang, Shiyuan Li, Jitong Chen, Xin Wei, Huizhan Lv and Xinhe Xu, 4*4 pattern and bayesian learning in Monte-Carlo go. In: *Proc. of the 13th International Conference on Advances in Computer Games, Revised Selected Papers,* (2011), Springer, 108-120.

[Yajima et al., 2010] Takayuki Yajima, Tsuyoshi Hashimoto, Toshiki Matsui, Junichi Hashimoto and Kristian Spoerer, Node-expansion operators for the UCT algorithm. In: *Proc. of the 7th International Conference on Computer and Games, Revised Selected Papers,* (2010), Springer,116-123.

Liite 1

Go-sanasto

Atari: Kivi tai kiviryhmä, jolla on vain yksi vapaa paikka [Sensei's Library, 2014].
Fuseki: Aloitussiirto [Sensei's Library, 2014].
Hane: Japanilainen termi, joka tarkoittaa siirtoa, joka ympäröi vastustajan kiven tai kiviryhmän [Sensei's Library].
Joseki: Tavallinen siirtosarja, joka yleensä pelataan nurkkaan [Müller, 2002]
Kivi: Pelinappula, joko valkoinen tai musta, joita asetetaan pelilaudalle
Komi: Tasoituspisteet, jotka annetaan valkoiselle pelaajalle pelin lopussa. Usein sisältää puolikkaan pisteen, jotta vältetään tasapelin mahdollisuus.
Ko-sääntö: Pelilaudan saman tilanteen toistaminen on kielletty. Ks. kuva 5
Kuollut kivi tai kiviryhmä: Kiviryhmä, jolla ei ole yhtään vapaata paikkaa. Nämä poistetaan laudalta. [Richards et al., 1998]
Piste: Pelilaudan viivojen risteymäkohtia, joihin kivet asetetaan.
Seki: Japanin kielestä lainattu termi kuvamaan tilannetta, jossa molemmilla pelaajilla on kaksi elävää kiviryhmää, joilla on yhteisiä vapaita paikkoja, mutta kumpikaan ei pysty täyttämään vapaita paikkoja ilman, että kiviryhmä kuolee. [Sensei's Library, 2014].
Semeai: Valloittamiskilpailu [Müller, 2002].
Silmä: Ympäröity alue, jossa on yksi vapaa paikka. Kivet, joilla on kaksi silmää, ei voida enää valloittaa [Müller, 2002].
Tikapuut (ladder): Alkeellinen valloittamissiirtosarja [Müller, 2002].
Vapaa paikka: Tyhjä paikka kiven tai kiviryhmän vieressä [Richards et al., 1998].