Outa Valkama

Go-tekoälyjen toteutusratkaisuja

Tietotekniikan
kandidaatintutkielma
7. syyskuuta 2009

JYVÄSKYLÄN YLIOPISTO
TIETOTEKNIIKAN LAITOS

Jyväskylä

Tekijä: Outa Valkama
Yhteystiedot: outa.j.valkama@jyu.fi
Työn nimi: Go-tekoälyjen toteutusratkaisuja
Title in English: AI Research in Go
Työ: Tietotekniikan kandidaatintutkielma
Sivumäärä: 26
Tiivistelmä: Go tarjoaa tekoälytutkijoille jatkuvasti uusia haasteita ja tapoja testata ideoitaan. Toistaiseksi ei ole vielä onnistuttu kehittämään tekoälyä, jota ihminen ei voisi voittaa, mutta yrityksiä on monenlaisia. Yksinkertaisista säännöistä huolimatta go on hyvin strateginen ja taktinen peli ja gon ratkaisemista pidetään yleisenä tavoitteena tekoälytutkimuksessa.
English abstract: Go provides constantly new challenges and ways to test ideas for AI researchers. So far there has been no success developing Go AI that human cannot beat, despite many tries. With simple rules, Go is very deep, stratetic and tactical game and succeeding in solving Go has often been considered as a breakthrough in AI research.
Avainsanat: Go, GNU Go, Fuego, tekoäly, tekoälymenetelmät
Keywords: Go, GNU Go, Fuego, AI, AI research

Sisältö

1 Johdanto

Go on toiminut tekoälytutkijoiden haasteena jo vuosia. Ohjelmistot olivat pelivahvuudeltaan pitkään hyvin alhaisella tasolla ja vasta viime vuosina on alettu löytää menetelmiä, jotka ovat vahvistaneet go-ohjelmistojen pelivahvuutta huomattavasti. Erilaisia menetelmiä gon tutkimisessa on hyvin monia.

Yleisin ongelma gon ratkaisussa on pelipuun nopea kasvaminen verrattuna esimerkiksi shakkiin. Eroja gon ja shakin välillä on käsitelty hieman enemmän luvussa 3. Muita haasteita gon pelaamisessa on ihmismäisen ajattelun tehokas mallintaminen tietokoneelle sekä esimerkiksi kuvioiden tunnistaminen laudalla.

Perinteisesti monessa pelissä käytetyt ns. brute force -menetelmät eivät toimi hyvin gossa edellä mainitun pelipuun kasvamisen takia. Näitäkin menetelmiä on yritetty kehittää golle sopiviksi kuten on tehty esimerkiksi Migos-ohjelmassa. Neuroverkkoja on käytetty varsin onnistuneesti esimerkiksi backgammonin pelaamisessa ja ne tuntuvat myös melko lupaavilta menetelmiltä gossa tietokoneiden laskentatehojen kasvaessa. Laskentatehojen rajoitteen takia neuroverkkoja on toistaiseksi vain tutkittu lähinnä 9x9-laudoilla. Muita käytettyjä tapoja ovat Monte-Carlo-menetelmät ja Monte-Carlo-puuhaut sekä pitkälti tietoon pohjautuvat menetelmät, joita hyödynnetään pitkälti GNU Go:ssa.

Luvussa 2 käsitellään hieman, mikä go on sekä käydään läpi gon säännöt. Luvussa 2.2 käydään hieman läpi, miksi gota tutkitaan. Luku 3 käsittelee eroja shakissa ja gossa. Luvussa 4 käydään läpi muutama eri menetelmä, joita on kokeiltu gon ratkaisemisessa. Luku 5 käsittelee GNU Go -ohjelmistoa ja luku 6 käsittelee Fuego-ohjelmistoa.

2 Mikä go on?

Go on yli kolmetuhatta vuotta vanha lautapeli, jonka epäillään saaneen alkunsa Kiinassa. Pelin säännöt ovat hyvin yksinkertaiset, mutta se tarjoaa paljon haastetta ja vaatii strategista silmää samoin kuin esimerkiksi shakki. Pelaajia pelissä on kaksi. Pelivälineenä toimivat yleensä 19x19-ruudun lauta sekä valkoiset ja mustat kivet, mutta myös pienempiä lautoja kuten 9x9 ja 13x13 käytetään usein varsinkin vastaalkajien keskuudessa. Peli aloitetaan aina tyhjältä laudata ellei pelissä ole tasoituskiviä, ja molemmat pelaajat pelaavat vuorotellen yhden kiven laudan ruutujen risteyskohtiin. Tarkoituksena on yrittää rajata laudalta alueita, ja pelaaja, jolla on pelin lopuksi enemmän aluetta, voittaa pelin.

2.1 Gon pelisäännöt

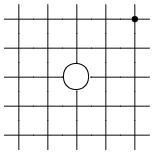
Pelivälineinä pelissä toimivat yleensä puusta tehty lauta, joka sisältää 19x19-kokoisen ruudukon. Ruudukon koko ei ole määrätty ja se voi olla mikä tahansa jaoton lukumäärä, mutta yleisin kokoluokka on 19x19. Pelinappuloina toimivat mustat ja valkoiset nappulat, joita kutsutaan kiviksi.

Pelin aloittaa aina mustilla kivillä pelaava pelaaja. Valkoinen aloittaa pelin vain jos pelissä käytetään tasoituskiviä. Tasoituskiviä käytetään yleensä silloin, jos toinen pelaajista on selkeästi vahvempi pelaaja kuin toinen. Tällöin laudalle asetetaan sovittu määrä mustan kiviä valmiiksi ja vahvempi pelaaja aloittaa pelin valkoisilla kivillä. Siirrot pelataan vuorotellen asettamalla yksi kivi laudalle tyhjään risteyskohtaan. Siirron voi myös jättää pelaamatta, jolloin siirtovuoro siirtyy vastustajalle ja tätä sanotaan passaukseksi.

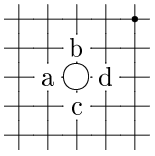
Kuva 2.1.1. Tyhjä kohta laudalta.

Kuva 2.1.2. Musta on pelannut siirron.

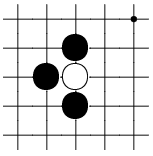
Gossa voidaan myös ”syödä” vastustajan kiviä viemällä niiden ”vapaudet” (katso kuvat 2.1.3-2.1.7). Vastustajan kivien syönti on tärkeä osa pelin pelaamista. Syödyt kivet lisätään vankeihin, jotka taas lasketaan mukaan pelin päätyttyä pistelaskussa. Jokainen vanki vastaa yhtä pistettä laudalla.



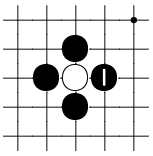
Kuva 2.1.3. Alkutilanne.



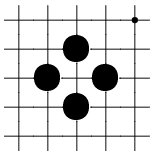
Kuva 2.1.4. Vapaudet.



Kuva 2.1.5. Atari.

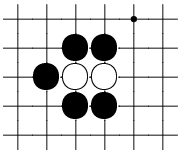


Kuva 2.1.6. Syönti.

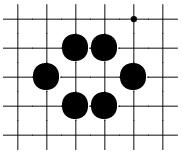


Kuva 2.1.7. Lopputulos.

Jos useampi samanvärinen kivi yhdistyy toisiinsa, muodostavat nämä ryhmän. Ryhmä voidaan syödä kerralla viemällä ryhmän viimeinen vapaus (katso kuvat 2.1.8 ja 2.1.9).

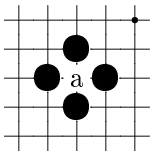


Kuva 2.1.8. Valkean ryhmällä on yksi vapaus.



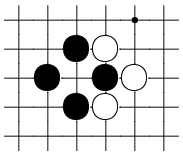
Kuva 2.1.9. Musta syö valkean ryhmän.

Gossa myös itsemurha on kielletty. Itsemurhalla tarkoitetaan, ettei pelaaja saa pelata pisteeseen, josta seuraisi oman ryhmän välitön laudalta poistaminen. Kuvassa 2.1.10 valkea ei voi pelata a:lla merkittyyn kohtaan, koska se olisi itsemurha. Valkea voisi pelata a:lla merkittyyn kohtaan vain jos se olisi mustan ryhmän viimeinen vapaus ja siihen pelaaminen johtaisi mustan ympäröivän ryhmän syömiseen.

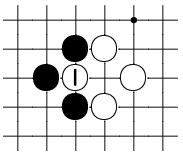


Kuva 2.1.10. Valkea ei voi pelata a:lla merkittyyn kohtaan.

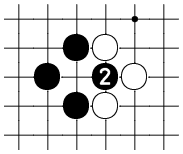
Gossa on myös ko-sääntö. Tämä sääntö tarkoittaa sitä, ettei edellinen laudan tilanne saa toistua välittömästi (katso kuvat 2.1.11, 2.1.12 ja 2.1.13).



Kuva 2.1.11. Alkutilanne.



Kuva 2.1.12. Laillinen siirto.



Kuva 2.1.13. Laiton siirto.

Peli päättyy kun molemmat pelaajat hyväksyvät, ettei pelattavia siirtoja ole enempää. Yleensä peli sovitaan päättyneeksi, kun molemmat pelaajat passaavat vuoronsa. Peli voi myös päättyä, jos toinen pelaaja hyväksyy olevansa selvästi tappiolla eikä näe syytä jatkaa peliä, jolloin hän luovuttaa pelin.

Pelin voittaa pelaaja, jolla on pelin päätyttyä enemmän aluetta hallussaan. Joissain säännöissä myös pelatut kivet lasketaan mukaan lopullisiin pisteisiin, mutta Suomessa pelatessa yleinen käytäntö on ollut vain laskea alueiden pisteet. Käytännössä sääntöjen eroavaisuudet eivät vaikuta pelistrategiaan.

2.2 Miksi gota tutkitaan?

Yleinen väärinkäsitys, joka johtaa juurensa vuosien shakin tutkimukseen on se, että ns. brute force -menetelmät pystyvät ratkaisemaan minkä tahansa formaalin ongelman kunhan ne toteuttavat hyvät haku- ja arviointialgoritmit [3, luku 1]. Go on toistuvasti todistanut, ettei sitä pystytä ratkaisemaan perinteisillä brute force -menetelmillä. Sen lisäksi, että gon pelipuu kasvaa huomattavasti nopeammin kuin esimerkiksi shakissa, myös pelistrategia ja taktiikka vaikuttavat paljon pelin lopputulokseen ja tämä tulisi ottaa huomioon myös go-ohjelmistoissa.

Shakissa laudan tilanne voidaan arvioida helposti käymällä läpi pelipuuta kunnes löydetään siirto, joka muuttaa tilannetta radikaalisti suuntaan tai toiseen. Gossa vastaava haku tuottaa vain taktisen tuloksen lokaalisti, mutta ei ota huomioon strategista pelaamista koko laudalla. Menetelmien tulisi siis ottaa huomioon myös koko laudan strateginen pelaaminen, mutta perinteisillä brute force -menetelmillä tämä ei toimi pelipuun kasvamisen takia. Brute force -menetelmät toimivat vielä hyvin pienillä laudoilla kuten 5x5-laudoilla (katso luku 4.1), mutta nämä vaativat myös tietokoneelta paljon laskentatehoa. Myös neuroverkkoja sekä Monte-Carlo-menetelmiä on tutkittu lisäten näihin joitain parannuksia (katso luvut 4.2 ja 4.3) toimimaan paremmin gon parissa. Toistaiseksi menestyksekkäimpänä menetelmänä on käytetty Monte-Carlo-puuhakuja eri ohjelmissa kuten esimerkiksi Fuegossa (katso luku 6).

Tietotekniseltä kannalta gon tutkiminen on tuottanut uusia mielenkiintoisia menetelmiä puiden läpikäynnissä ja arvioinnissa sekä tuonut parannuksia jo olemassa

oleviin menetelmiin. Pelipuun koon takia sen tehokas karsinta vaatii optimointeja algoritmeihin ja tutkimaan mahdollisia uusia lähestymistapoja. Lisäksi go toimii melko hyvänä tapana testata hakualgoritmeja ja uusia ideoita tekoälyissä käytännössä. Muita tieteenaloja gon tutkiminen kiinnostaa siinä mielessä, että tehokas gotekoälyn tekeminen vaatii osittaista ihmisen ajattelutavan mallintamista matemaattisesti, jotta tietokone voi tehokkaasti päätyä samaan lopputulokseen mihin ihminenkin.

3 Shakin ja gon eroja

Shakki on läntisessä kulttuurissa melko suosittu lautapeli ja on vaikeaa löytää henkilöä, joka ei olisi kuullut pelistä tai ei osaisi sen sääntöjä edes jollain tasolla. Shakkien tekoälyt ovat jo mestaritasolla ja kaikki tekoälyistä kiinnostuneet tietävät varmasti Deep Blue -supertietokoneen, joka voitti Garry Kasparovin, shakin maailmanmestarin. Sen sijaan go on yksinkertaisemmista säännöistä huolimatta vieläkin täysin ratkaisematta. Gota ja shakkia voidaan helposti vertailla kahdellatoista eri ominaisuudella [3, luku 2.2].

1. Erityyppisiä pelinappuloita on vähemmän gossa. Shakissa näitä on 6 erilaista, kun taas gossa niitä on vain 1.

2. Lautojen koot ovat hyvin erikokoisia. Shakin laudan koko on 8x8 neliötä, kun taas gossa laudan koko on yleensä 19x19-ruudukko. Tästä johtuen keskimääräinen siirtomäärä shakissa on noin 80, kun taas gossa vastaava on noin 300.

3. Gossa siirron voi laittaa mihin tahansa laudalla (poislukien itsemurhatilanteet, katso 2.1), kun taas shakissa siirron laillisuutta rajoittaa itse pelinappula sekä laudan sen hetkinen tilanne. Tästä syystä gossa keskimääräinen haarautuminen pelipuussa jokaiselle siirrolle on hyvin suuri (arvioitu noin 200) verrattuna shakkiin (arvioitu noin 35).

4. Shakissa pelin päättyminen on helppo tarkistaa: shakki-matti tarkistetaan vain sen hetkisen laudan tilanteen ja kuningasta uhkaavien pelinappuloiden määrästä. Gossa sen sijaan peli päättyy vasta kun molemmat pelaajat passaavat ja peliin saatetaan vielä palata, jos jokin tilanne on epäselvä. Amatöörien on monesti vaikea tunnistaa, milloin peli on jo ohi ja pitkittävät peliä siinä missä parempi pelaaja olisi jo lopettanut pelin. Go-ohjelmistoilla on myös monesti vaikeuksia nähdä, milloin peli on jo ohi ja pitkittävät turhaan peliä ja tämä kasvattaa myös pelipuuta.

5. Sekä shakissa että gossa siirrolla voi olla hyvinkin kauaskantoisia vaikutuksia. Shakissa nämä vaikutukset tulevat suoraan pelinappuloiden tavasta liikkua laudalla, kun taas gossa yksittäinen siirto voi pelata merkittävää roolia vasta myöhemmin esimerkiksi ryhmän vangitsemisessa.

6. Shakissa laudan tilanne muuttuu nopeasti aina pelinappuloiden liikkuessa. Gossa vastaava laudan muutos on enemmänkin asteittainen, koska kiviä lisätään aina olemassaolevaan tilanteeseen. Gossa suuret laudan muutokset tapahtuvat yleensä, kun vastustajan ryhmät ”kuolevat” tai syödään pois laudalta. Gossa laudan muu-

tokset vaativat aina enemmän muistia, kun taas shakissa muistia ei tarvita läheskään niin paljoa pelinappuloiden vähentyessä laudalta.

7. Shakissa laudan tilanteen arviointi koneellisesti on suhteellisen helppoa. Jokainen pelinappula voidaan arvottaa tietyn arvoiseksi ja arvioimalla laudan tilannetta ja ottamalla huomioon eri pelinappulat voidaan melko helposti päätellä todennäköisempi voittaja. Gossa vastaava on hankalampaa, koska kivet ovat aina samanarvoisia oli laudan tilanne mikä tahansa. Vaikka gossa on tarkoituksena vallata aluetta ja näin saada pisteitä, peli yleensä aloitetaan rakentamalla vaikutusvaltaa ja aikainen alueen varmistaminen katsotaan ylipelaamiseksi, jonka seurauksena on yleensä hyvin isojen pistemäärien menettäminen. Go-ohjelmistoille on yleensä hankalaa määrittää, miten vaikutusvaltaa rakennetaan ja miten se hyödynnetään jälkeenpäin varmistetuksi alueeksi.

8. Edellä mainittujen syiden takia yleisin tapa rakentaa shakki-tekoälyä on käyttää vahvasti hyödykseen puuhakuja sekä jotain sopivaa arviointimenetelmää. Vastaavat menetelmät eivät suoraan toimi gossa, koska pelipuu laajenee eksponentiaalisesti pelin edetessä ja pelipuun kasvu hidastuu reilusti vasta keskipelin loppupuolella, jolloin järkevien siirtojen määrä vähenee. Tästä syystä perinteiset brute force -menetelmät pelipuun läpikäyntiin eivät sovellu gon kanssa toisin kuin shakissa. Pelipuun karsinta on myös hyvin hankalaa gossa ilman erittäin hyvää arviointimenetelmää.

9. Shakin ja gon pelipuiden hauissa on suuria eroja tehtäessä taktisia arviointeja. Molemmissa peleissä ihmiset oppivat nopeasti lukemaan useampia siirtoja eteenpäin, mutta gossa esimerkiksi melko yleinen ”tikapuu”-kuvio voi johtaa jopa kuudenkymmenen siirron päähän lukemista. Toisaalta ihminen oppii nopeasti näkemään nopealla vilkaisulla, miten esimerkiksi nämä ”tikapuut” päättyvät ja osaa toimia sen mukaan. Go-ohjelmistoille ”tikapuiden” osittainen laskeminen ei hyödytä, koska pitemmälle pelaamalla voi tulla vastaan kivi, joka muuttaa tilanteen kokonaan. Sekä shakissa että gossa ammattilaiset lukevat usein monia erilaisia siirtosekvenssejä läpi, mutta shakissa sekvenssi on harvoin yli kymmenen siirtoa pitkä toisin kuin gossa.

10. Horisonttiefekti heuristisessa haussa ilmenee, kun haku tietylle syvyydelle palauttaa arvion tilanteesta, joka voi olla hyvin erilainen verrattuna arvioon, joka olisi tehty mentäessä hieman syvemmälle pelissä. Shakissa tämä on enemmänkin ongelma vasta ammattitason peleissä, mutta gossa tämä tulee vastaan jo aloittelijoidenkin kohdalla kuten aikaisemmin mainitussa ”tikapuu”-kuviossa.

11. Toisin kuin shakissa, gossa on tärkeää tunnistaa, mihin ryhmiin pienemmät ryhmät tai yksittäiset kivet kytkeytyvät, jotta näiden status ja tehokkuus voidaan

määritellä. Go-ohjelmilla on yleensä ongelmia tunnistaa, mihin ryhmään tietyt kivet yhdistyvät ja näin ollen ohjelmien on yleensä hankala määrittää kivien tehokkuus ja vaikutusvalta.

12. Yksi suurimmista eroista shakissa ja gossa tulee itse pelaajista. Shakissa vaaditaan keskimäärin samantasoiset pelaajat, jotta peli olisi tasaista, kun taas gossa on hyvin tehokas avustusmekanismi, jonka avulla eritasoiset pelaajat voivat pelata tasaisia pelejä. Go-ohjelmistojen ongelmana ihmistä vastaan pelatessa on se, että ihminen voi oppia nopeasti tekoälyn rajoitteet toisin kuin tekoäly ihmisen rajoitteet. Lisäksi tekoäly pysyy muuttumattomana pelin läpi, kun taas ihminen voi muuttaa pelityyliänsä kesken pelinkin.

4 Erilaisia menetelmiä

Go on todistetusti ollut vaikea ratkaista tietokoneella. Perinteisten ratkaisumenetelmien ollessa hyödyttömiä gota ratkaistaessa on alettu etsiä uusia ratkaisutapoja.

4.1 Migos

Migos oli ensimmäinen go-tekoäly, joka ratkaisi 5x5-laudan ja tätä pidettiinkin melkoisena edistysaskeleena tekoälyissä gossa [12]. Migos käyttää hyväkseen heuristista arviointifunktiota, jolla on viisi päämäärää: (1) maksimoida kivien määrä laudalla, (2) maksimoida vapauksien määrä kiville, (3) välttää siirtoja laudan reunalla, (4) kytkeä kivet ja (5) rakentaa ryhmille ”silmiä”, jotta ryhmä elää [12, luku 3.1]. Vastaavasti samat päämäärät ovat päinvastaisena vastustajan kiville.

Pelipuun läpikäynnissä Migos käyttää hyväkseen alpha-beta-haun variaatiota, iteratiivisesti syventyvää Principal Variation Search -hakua (PVS) [12, luku 4]. Tähän hakuun on lisätty vielä pari parannusta: (1) vuorottelutaulut (transposition tables), (2) symmetriahaut (symmetry lookups), (3) internal unconditional bounds (ei suomennosta) ja (4) paranneltu siirtojärjestys (enhanced move ordering). Vuorottelutaulujen ideana on lähinnä estää tilanteet, joissa etsitään samaa tilannetta useaan kertaan pelipuusta laittamalla ylös aiemmin vastaan tullut paras siirto, pistemäärä ja syvyys puussa [12, luku 4.1]. Symmetriahaulla taas pyritään pienentämään muistinkäyttöä, koska tallennettava pelitilan määrä vähenee symmetrian myötä. Tärkeimpänä sama pelitilanne, jossa on mustan vuoro pelata, voidaan esittää myös valkean pelitilanteena kun värit muutetaan käänteisiksi. Näin samaa pelitilannetta ei tarvitse pitää muistissa molemmille pelaajille erikseen [12, luku 4.2].

Vaikka Migos pystyy ratkaisemaan tyhjän 5x5-laudan, ei laskentateho riittänyt vielä Migoksen julkaisun aikaan ratkaisemaan tyhjää 6x6-lautaa. Nykyisillä moniydinkoneilla Migos voisi onnistua jo paremmin, mutta Monte-Carlo-puuhaut tuntuvat tuottavan paljon parempia tuloksia (katso luku 6).

4.2 Neuroverkot

Yksi ratkaisumenetelmä go-ohjelmistoissa on Miikkulaisen ja Lubbertsin esittämä tapa käyttää kehittyviä neuroverkkoja [10]. Näissä neuroverkoissa on käytetty hyödyksi SANE (Symbiotic Adaptive Neuro-Evolution) -metodia kehittämään verkkoja gon pelaamisessa pienillä laudoilla ilman esiohjelmoituja tietoja itse pelistä (poislukien pelin säännöt). Neuroverkkoja on käytetty ennenkin hyödyksi gon ratkaisemiseen, mutta ongelmaksi tuli taso, jolle neuroverkko pystyi kehittymään [10, luku 3]. Tuolloin neuroverkon opetustaso määräytyi pitkälti vastustajan tasosta. Kun vastustajan taso oli ylitetty, ei neuroverkko voinut enää kehittyä pidemmälle. Tämän takia Miikkulainen ja Lubberts päättivät kokeilla samaa ideaa kahdella erillisellä kehittyvällä neuroverkolla, jotka pelaavat toisiaan vastaan. Näin teoreettisesti neuroverkon tulisi kehittyä koko ajan.

SANE eroaa normaaleista kehittyvistä neuroverkoista hieman. Sen sijaan, että koko neuroverkko kehittyisi kuten perinteisissä menetelmissä, SANEssa kehittyy joukko neuroneita sekä suunnitelmia tai kaavoja, missä kaavat kertovat, mitkä neuronit kytkeytyvät itse neuroverkkoon [10, luku 3.2]. Kehittämällä neuroneita koko verkon sijaan, hakuavaruus voidaan hajoittaa osiin ja näin ryhmä neuroneita voi keskittyä spesifiseen osaan tehtävässä. Näin vältytään myös algoritmin jumittuminen suboptimaaliseen ratkaisuun yhtä helposti kuin muilla neuroverkoilla.

Yksi iteraatio SANEn evoluutiossa on jaettu kahteen osaan: arviointivaiheeseen ja lisääntymiseen. Arviointivaiheessa jokaisella kaavalla rakennetaan yksi neuroverkko ja tämän jälkeen verkkoa arvioidaan. Arvoinnin päättyessä verkoille annetaan oma sopivuusarvo. Jokaiselle neuronille määrätään myös sopivuusarvo, joka määräytyy summana viiden parhaan verkon sopivuusarvoista, joihin neuroni on kytketty. Lisääntymisvaiheessa molemmat populaatiot arvotetaan sopivuusarvojen mukaan. Jokaiselle eliittiyksilölle valitaan pari, joka myös kuuluu eliitteihin, ja näiden avulla muodostetaan kaksi jälkeläistä. Nämä jälkeläiset korvaavat huonoimmin menestyneet yksilöt populaatiossa. Lisääntymisvaiheessa on myös 0.5% mahdollisuus muodostua mutaatio sekä neuronien että kaavojen populaatioissa.

Koevoluutiolla tarkoitetaan kahden tai useamman populaation yhtäaikaista evoluutiota yhteisellä sopivuusnäkyvyydellä. Kilpailullinen koevoluutio voidaan määritellä isäntäpopulaation evoluutiolla, missä yksilöt kilpailevat suoraan parasiittipopulaation, joka myös kehittyy, yksilöitä vastaan [10, luku 3.3]. Miikkulainen ja Lubberts käyttävät hieman eri määritelmää isännistä ja parasiiteistä [10, luku 3.4]. Isäntä on yksilö, joka pyrkii optimaaliseen ratkaisuun, kun taas parasiitti pyrkii voittamaan isännän käyttämällä hyväkseen tämän heikkouksia ja tätä on yritetty myös

mallintaa SANEssa.

Kilpailullisen koevoluution tekniikkoina voidaan pitää kilpailullista sopivuuden jakamista, jaettua näytteenottoa ja kunniagalleriaa. Miikkulainen ja Lubberts käyttivät myös samoja tekniikoita kilpailullisen koevoluution toteuttamiseen [10, luku 3.3]. Kilpailullisen sopivuuden jakamisen seurauksena ainoat yksilöt, jotka ”voittavat” tietyt vastustajat, säästyvät sukupuutolta, vaikka nämä muuten olisivatkin huonompia, koska näille yksilöille jaetaan suurempia ”palkintoja” voitostaan [10, luku 3.3.1]. Näin ollen monipuolisuutta populaatiossa suositaan ja näiden yksilöiden tieto jakautuu populaatiolle, jolloin jälkeläisetkin voivat voittaa samat vastustajat. Kunniagallerian tavoitteena on varmistaa, että yksilöt eivät menetä kykyään voittaa edellisiä generaatioita [10, luku 3.3.3]. Kunniagalleria pitää yllä parasta vastustajaa joka generaatiolta. Näin jokaisen yksilön täytyy pystyä myös voittamaan satunnainen joukko kunniagalleriasta sen lisäksi, että yksilö voittaa nykyisen generaation.

Verkko, jota lopulta käytettiin testeissä, koostui 2000 neuronin populaatiosta sekä 200 kaavan populaatiosta, jonka lisäksi piilotetun kerroksen kokona oli 100 neuronia [10, luku 4.1]. Sisääntulokerros koostuu kahdesta syöttöyksiköstä jokaista leikkauspistettä kohti laudalla. Ensimmäinen syöttöyksikkö aktivoidaan, jos mustan kivi pelataan kyseiseen kohtaan ja toinen, jos pelattu kivi on valkoinen. Ulostulokerros koostuu taas yhdestä yksiköstä leikkauspistettä kohti ja voi saada arvon 0 ja 1 väliltä. Arvo kertoo, miten kannattavaa kyseiseen pisteeseen on pelata, ja suurempi arvo tarkoittaa aina kannattavampaa. Verkko päättelee seuraavan siirtonsa antamalla sisääntulokerroksen syötteeksi nykyinen laudan tilanne ja pelattavaksi siirroksi päätyy siirto pisteeseen, joka sai suurimman arvon ulostulokerroksessa. Jos kaikki arvot ovat 0.5, neuroverkko passaa siirron.

Miikkulainen ja Lubberts muodostivat testeissään 250 generaatiota neuroverkosta [10, luku 4.2]. Jokaisella generaatiolla paras isäntä ja paras parasiitti tallennettiin. Generaatioiden muodostamisen jälkeen pidettiin turnajaiset. Jokainen isäntä tallennettujen verkkojen populaatiosta laitettiin pelaamaan jokaista parasiittia vastaan ja jokaisen isännän voittotulos tallennettiin. Mielenkiintoisesti verkot tuntuivat oppivan elävien ryhmien tekemisen. Vaikka suurin osa verkoista pelasi myös omalle alueelleen, suurin osa myös lopetti omalle alueellensa pelaamisen ajoissa ettei ryhmä kuollut.

Tärkeimpänä huomiona koevoluutio todella parantaa pelaamista [10, luku 5.1]. Koevoluutiolla kehittyneiltä verkoilta meni viisi generaatiota voittaa GNU Go, kun taas verkoilta, jotka käyttivät GNU Go:ta opetusvastuksena, voittaminen vei kahdeksantoista generaatiota. On tosin huomioitavaa, ettei GNU Go:ta ole todennäköi-

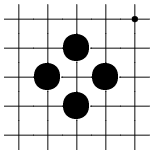
sesti kehitetty pelaamaan tehokkaasti pienillä laudoilla ja on muutenkin hankala sanoa, miten hyvin neuroverkot toimivat kokonaisella 19x19-laudalla.

4.3 Monte-Carlo-menetelmä

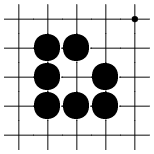
Toinen tapa on käyttää hyväkseen tunnettua Monte-Carlo-menetelmää. Bouzy ja Helmstetter rakensivat kaksi go-ohjelmaa, OLGA ja OLEG, testaamaan menetelmää nykykoneiden laskentatehoilla [2]. Perusideana on, että jokaiselle siirrolle pelataan annettu määrä täysin satunnaisia pelejä loppuun asti ja lasketaan lopputuloksen pisteet. Pelattavaksi siirroksi päätyy siirto, joka tuottaa suurimman pistemäärän [2, luku 3].

Bouzy kehitti OLGAn jatkona INDIGOn kehitykselle ja pääasiallisena ideana oli käyttää lähestymistapana mahdollisimman vähän tila riippuvaista tietoa [2, luku 3.2]. Helmstetter kirjoitti taas OLEGin. OLEGin lähtökohtana oli kehittää go-ohjelma, jossa olisi mahdollisimman vähän varsinaista tietoa itse go-pelistä [2, luku 3.2]. Molempien ohjelmistojen pelin laatu riippuu odotetusta tarkkuudesta, joka vaihtelee tehtyjen testien määrän mukaan. Aika, joka menee tehdä näitä testejä, on verrannollinen aikaan, joka menee pelata yksi satunnainen peli. Käytettäessä 2GHz:n tietokonetta OLGA pelaa noin 7000 peliä sekunnissa ja OLEG taas noin 10000 peliä sekunnissa.

OLGA ja OLEG ovat hyvin pitkälti samankaltaisia, mutta niissä on yksi oleellinen ero: molemmat määrittelevät silmämuodon ryhmälle eri tavalla. Silmämuodon määritteleminen on hyvin tärkeää Monte-Carlo-pohjaisessa go-ohjelmistossa, koska ilman tätä määritelmää ohjelma ei saisi koskaan elävää ryhmää laudalle ja peli jatkuisi ikuisesti [2, luku 3.3]. OLGA määrittelee silmämuodon tyhjänä risteyskohtana, jota ympäröi saman väriset kivet ja näillä kivillä tulee olla kaksi tai enemmän vapauksia (katso Kuva 4.3.1). OLEG taas määrittelee silmämuodon tyhjänä risteyskohtana, jota ympäröi saman väriset kivet ja näiden kivien tulee kuulua samaan ryhmään (katso Kuva 4.3.2). Tavoitteena on ollut määritellä silmämuoto mahdollisimman yksinkertaisesti ja nopeasti laskettavasti. Haittapuolena molemmat määritelmät voivat myös olla väärässä tietyissä tilanteissa.



Kuva 4.3.1. OLGAn määritelmä silmämuodosta.

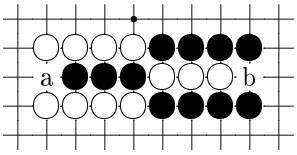


Kuva 4.3.2. OLEGin määritelmä silmämuodosta.

Bouzy ja Helmstetter toteuttivat useita mahdollisia parannuksia OLGAan ja OLEGiin ja testasivat, miten nämä mahdollisesti parantavat peliä [2, luku 3.4]. Nämä mahdolliset parannukset ovat progressive pruning, simulated annealing, all-moves-as-first-heuristiikka ja minimax-haku syvyydellä 2.

Progressiivisessa karsinnassa (progressive pruning) siirto karsitaan heti, jos tarpeeksi monen pelatun pelin jälkeen (testeissä käytettiin 100 peliä siirtoa kohti) siirto todetaan tilastollisesti huonommaksi kuin jokin toinen siirto [2, luku 4.1]. Karsintaprosessi lopetetaan, jos siirtoja on jäljellä enää yksi, jos kaikki jäljellä olevat siirrot todetaan yhtä hyviksi tai jos ennaltamäärätty raja-arvo iteraatioita saavutetaan. Karsinnan hyötynä on nopeus. Mitä enemmän siirtoja karsitaan, sitä nopeampaa pelien läpikäynti on, mutta toisaalta tämä voi myös heikentää peliä.

Kun loppuun pelattua satunnaista peliä arvioidaan, voidaan tämä lopputilanne muodostaa useasta eri satunnaisesta pelistä, missä ensimmäinen siirto sekä satunnainen toinen siirto ovat voineet vaihtaa paikkoja. Näin ollen, kun lopputulosta arvioidaan, voidaan tulos määritellä vain pelin ensimmäiselle siirrolle tai ajatella jokaisen pelatun siirron olleen ensimmäinen siirto pelissä ja määritellä näin tulos jokaiselle siirrolle. Bouzy ja Helmstetter kutsuvat jälkimmäistä all-moves-as-first-heuristiikaksi [2, luku 4.2]. Heuristiikka nopeuttaa Monte-Carlo-menetelmiä huomattavasti ja toimii hyvin tilanteissa, missä siirtojärjestyksellä ei ole varsinaisesti väliä [2, luku 4.2.2]. Toisaalta jos siirtojärjestyksellä on jotain väliä, heuristiikka voi johtaa selkeästi huonompaan tulokseen (katso kuva 4.3.3).



Kuva 4.3.3. Siirtojärjestyksellä on merkitystä.

Simuloitua lämpökäsittelyä (simulated annealing) ei todettu kovin hyödylliseksi menetelmäksi [2, luku 4.4]. Erot simuloidussa ja vakioksi asetetun lämpötilan eroissa olivat keskimäärin 1.6 pistettä simuloidun hyväksi. Ero ei ole tarpeeksi suuri, jotta nopeutta kannattaisi uhrata simuloidun version hyväksi.

Mielenkiintoisesti minimax-haku syvyydellä 2 tuottaa odotusten vastaisesti huonompia tuloksia kuin ilman tätä ”parannusta” [2, luku 4.5]. Tämän lisäksi menetelmä myös hidastaa ohjelmaa huomattavasti. Syytä tähän selvittäessä kävi ilmi, että syvyyden 1 minimax-haku yliarvioi juurisolmun minimax-arvon, kun taas syvyyden 2 minimax-haku aliarvioi juurisolmun minimax-arvon. Tämän johdosta OLGA, joka käyttää syvyyden 2 hakua passaa siirron joissain tilanteissa, missä syvyyden 1 hakua käyttävä OLGA taas ei.

Testatakseen Monte-Carlo-menetelmien tehokkuutta, Bouzy ja Helmstetter suorittivat pienen turnajaisen, missä Monte-Carlo-ohjelmistot pelasivat tietoon perustuvia ohjelmistoja kuten GNU Go:ta vastaan [2, luku 4.6]. Pelit pelattiin 9x9-laudalla ja odotusten mukaisesti GNU Go voitti jokaisen pelin. Piste-ero oli keskimäärin yli 30 pistettä GNU Go:n hyväksi, mutta ottaen huomioon siirtojen generoinnin kompleksisuuden eron GNU Go:ssa ja OLGAssa sekä OLEGissa, tulos on varsin tyydyttävä.

Monte-Carlo-menetelmien suurimpana etuna on niiden suhteellisen helppo toteuttaminen ohjelmoinnin näkökulmasta. Ainoa varsinainen go-sääntö, joka täytyy toteuttaa, on silmämuodon määrittäminen. Lisäksi Monte-Carlo-menetelmillä vältetään pelin pilkkominen pienempiin osapeleihin, tapa jota useat tietoon perustuvat ohjelmistot käyttävät helpottaakseen laskemista. Lisäksi arviointi suoritetaan loppuun pelatusta pelistä, jolloin arviointi on triviaalia. Ihmisnäkökulmasta Monte-Carlo-ohjelmistot tuntuvat aliarvioivan molemmat osapuolet. OLEG ja OLGA tekevät usein helposti vahvoja kytkettyjä muotoja sekä yrittävät hyödyntää heikkouksia vastustajissa, mitä ei todellisuudessa ole.

OLEG ja OLGA hyötyvät selkeästi progressiivisesta karsinnasta sekä all-moves-as-first-heuristiikasta nopeuttamalla molempien peliä ilman, että ohjelmat menettävät paljoa varsinaista vahvuuttaan pelata peliä [2, luku 6]. Lisäksi vakiona asetettu ”lämpötila” parantaa pelin tasoa, mutta hidastaa hieman itse ohjelmaa. Simuloitu lämpökäsittely taas tekee ohjelmasta turhan monimutkaisen ja hidastaa sitä huomattavasti eikä paranna pelin laatua merkittävästi. Minimax-haku syvyydellä 2 ei myöskään parantanut pelin laatua.

Monte-Carlo-ohjelmistot paranevat helposti konetehojen kasvaessa. Jo kasvattamalla pelattujen pelien määrää siirtoa kohti saadaan aikaan parempia tuloksia. Myös esimerkiksi taktiikan lisääminen ja menetelmän muuttaminen osittain tietoon perustuvaksi parantavat oletettavasti pelien laatua [2, luku 6].

5 GNU Go

GNU Go on vapaan lähdekoodin projekti ja kaikkien asiasta kiinnostuvien saatavilla projektin kotisivuilta [8]. Seuraavassa tarkastellaan hieman tarkemmin itse ohjelman toimintaa.

GNU Go:n toiminta yrittäessä ratkaista, minne seuraava siirto pelataan, jakautuu pitkälti kolmeen vaiheeseen: tiedon keruuseen, mahdollisten siirtokandidaattien generointiin ja ”parhaan” siirron valintaan [9, luku 4].

Tiedon kerääminen aloitetaan tutkimalla tilannetta ja luomalla ”matoja”. Madot ovat suoraan toisiinsa yhdistyneitä kiviä ja näille lasketaan kivien määrä madossa sekä madon vapaudet. Seuraavaksi kutsutaan taktisen laskennan koodia jokaiselle madolle laudalla ja yritetään päätellä, onko jokin ryhmä mahdollista syödä vai ei. Kun tiedetään matojen sen hetkinen tilanne, voidaan yrittää piirtää ensimmäinen kuva laudan tilanteesta ja laskea matojen vaikutusvaltaa laudalla. Seuraava vaihe on päätellä, mitkä madot pystytään leikkaamaan toisistaan erilleen. Tämä päättely tehdään pääasiassa vertaamalla valmiisiin malleihin. Näille lasketaan sitten raaka arvio ryhmän silmien määrästä, katsotaan mitkä madot mahdollisesti muodostavat varmaa aluetta ja mitkä muodostavat potentiaalista aluetta sekä tehdään arvaus potentiaaliselle karkaamiselle, jos mato on hyökkäyksen alla. Jos jokin madoista katsotaan ”heikoksi”, tehdään sille ”elämä ja kuolema” -analyysi. Tämän jälkeen vaikutusvalta-analyysi tehdään uudestaan tekemään tarkempi arvio mahdollisesta alueesta. Aluearvion mahdolliset virheet pyritään korjaamaan niin sanotulla break-in-moduulilla, joka tarkoituksella laskee sekvenssejä, joilla yritetään tunkeutua alueen sisään [9, luku 4.1].

Tiedon keruun jälkeen aloitetaan seuraavan siirron mahdollisten kandidaattien generoiminen. Näitä kandidaatteja saadaan ulos erilaisista moduuleista, jotka laskevat oikeutuksia tietylle siirrolle. Tällaisia kandidaatteja ovat esimerkiksi siirrot, jotka kaappaavat tai puolustavat matoja, ja siirrot, jotka hyökkäävät vastustajan alueen sisään tai pienentävät sitä. Tärkeitä kandidaatteja ovat myös fuseki-siirrot eli alkupelin siirrot ja muotosiirrot. Muotosiirrot ovat yksi tärkeimmistä, koska se etsii tietokannoista mahdollisia siirtoja tiettyihin vakiintuneisiin muotoihin pelissä sekä joseki-tietokannasta joseki-siirtoja [9, luku 4.2]. Joseki on standartoitunut siirtosekvenssi, jonka katsotaan antavan suunnilleen samanarvoisen lopputuloksen lokaalisti molemmille pelaajille.

Kun kaikki kandidaatit on luotu, aloitetaan arviointi seuraavasta siirrosta kandidaattien pohjalta. Jokainen ehdotettu kandidaatti analysoidaan tarkasti ja siirrolle annetaan tietty painoarvo. Tärkein painoarvo siirrossa on sen alueellinen vaikutus. Muita vastaavia arvoja ovat siirron strateginen arvo, muodollinen arvo ja toissijainen arvo [9, luku 6.4]. Lopulliseksi siirroksi valitaan kandidaatti, joka on saanut suurimman painoarvon arvioinnin päätteeksi [9, luku 4.3].

6 Fuego

GNU Go:n lisäksi löytyy myös muita avoimen lähdekoodin projekteja ja näistä yksi on Fuego [6]. Fuego on kaikkien saatavilla projektin kotisivuilta [6] ja sen dokumentaatio löytyy myös WWW-muodossa [7]. Fuego siirtyi avoimen lähdekoodin alle vuonna 2008. Fuego ei rajoitu vain gon pelaamiseen vaan se toimii enemmänkin geneerisenä alustana useammalle lautapelille. Tässä tutkielmassa keskitytään kuitenkin vain Fuegon gon osaamiseen. Fuegon arvioidaan yltävän parhaillaan dan-tason pelaamiseen, kun GNU Go pysyy noin 8 kyun tasolla. Pelaajat luokitellaan käyttäen dan- ja kyu-arvoja. Vasta-aloittelija on yleensä 30 kyu ja voi nousta aina 1 kyu-arvoon asti. Tämän jälkeen luokitus nousee 1 daniin ja voi nousta aina 9 daniin asti. GNU Go:n perustuessa enemmän menetelmiin ennen Monte-Carlo-menetelmien yleistymistä go-ohjelmistoissa, Fuego käyttää vahvasti hyväkseen Monte-Carlo-puuhakua (Monte-Carlo Tree Search, MCTS). MCTS-menetelmät ovat vieneet go-ohjelmistoja eteenpäin viimeisen neljän vuoden aikana hyvin paljon. GNU Go:ssa on nykyään olemassa myös jonkinlainen toteutus MCTS-menetelmien hyväksikäyttöön, mutta se ei yllä suorituskyvyllään lähellekään Fuegoa. Go-ohjelmistot ovat nykyisellään lähellä ratkaista 7x7-laudan pelejä MCTS-menetelmien takia ja ovat saavuttaneet jo amatöörien dan-tason 9x9-laudoilla. Kaikkein parhaimmat ohjelmistot yltävät jo dan-tason peleihin jopa 19x19-laudoilla. American Go Association uutisoi 8.7.2008 aiheesta, kun MoGo voitti Myungwan Kimin, 8 dan pro -pelaajan, 9-kiven tasoituspelissä MoGon pelatessa mustaa [1].

Fuego perustuu kahteen aikaisempaan projektiin: Kierulfin Smart Game Boards ja Müllerin Exploreriin [5, luku 2.1]. Smart Game Boards on työpöytä tai pohja pelejä pelaaville ohjelmistoille ja Explorer taas gota pelaava ohjelma, joka on rakennettu Smart Game Boardsin päälle.

Fuegon pohjan muodostavat GtpEngine-kirjasto sekä SmartGame-kirjasto. GtpEngine tarjoaa peliriippumattoman toteutuksen Go Text Protokollasta eli GTP:stä. Yksinkertaisimmillaan moottori ajaa komentosilmukan, parsii tästä komennot ja kutsuu komentojen käsittelijää [5, luku 2.3]. Tämän lisäksi GtpEngine tukee myös ”pohdintaa”, joka antaa moottorille mahdollisuuden ajatella vastustajan siirtojen aikana. Perusmoottorista löytyy myös ilmeisesti kattava tuki moniajolle, jota Fuego käyttää hyväkseen.

SmartGame-kirjasto tarjoaa toteutukset peliriippumattomista yleisesti käytetyis-

tä funktioista ja luokista pelitilanteen tallentamisesta kahden pelaajan lautapeleissä kuten pelipuun lataamiseen ja tallentamiseen [5, luku 2.4]. Lisäksi SmartGame sisältää peliriippumattomat toteutukset alpha-beta-haulle sekä Monte-Carlo-puuhaulle. Perus MCTS-luokka SmartGame-kirjastossa implementoi UCT-haun [4] lisäten siihen muutaman yleisesti käytetyn parannuksen [5, luku 2.4.2].

Go-kirjasto on rakennettu SmartGamen päälle ja sisältää kaiken go-spesifisen tiedon. Go-kirjasto sisältää esimerkiksi tehokkaan implementoinnin go-laudasta, joka pitää yllä siirtohistoriaa ja antaa mahdollisuuden perua siirtoja sekä luokan go-spesifisille GTP-komennoille [5, luku 2.5]. Lisäksi go-kirjasto sisältää toteutukset ryhmien elämisen tarkistamiseen. Lisäksi Fuego sisältää go-ohjelmistoille yleisen ”Opening Book”-luokan, jonka tarkoituksena on parantaa ohjelman aloitussiirtoja pelissä [5, luku 2.5.1]. Fuego sisältää erillisen kirjan 9x9-laudoille sekä 19x19-laudoille. Yksinkertaisuuden vuoksi kirja sisältää vain yhden sekvenssin jokaiselle variaatiolle. Tällaisella yksinkertaistamisella on varjopuolensa varsinkin ihmisvastustajien kanssa, koska ihmiset eivät seuraa sekvenssejä aina kovin orjallisesti ja monet pelaajat pelaavat mielellään aina monimutkaisimman sekvenssin.

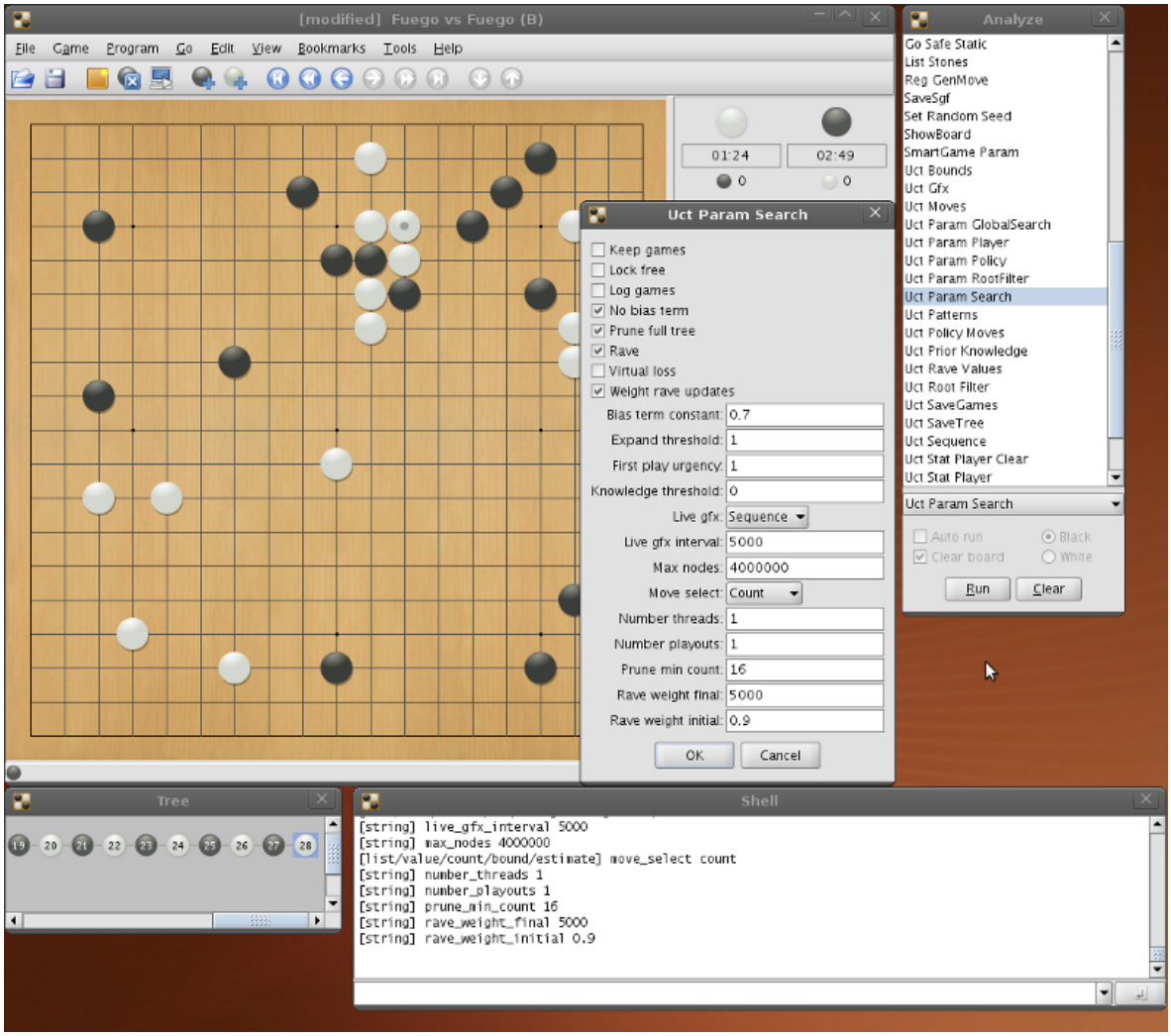
Fuegon kirja on ihmisen kirjoittama, kun taas esimerkiksi MoGon kirja on generoitu koneellisesti. Fuegon kirjat sisältävät lähinnä yleisesti käytettyjä aloitussekvenssejä, mutta mielenkiintoisena ratkaisuna Fuego ei sisällä minkäänlaista josekikirjaa. Syy joseki-kirjan poisjättämiseen olisi mielenkiintoista tietää, koska oletettavasti ihminen, joka tietää suuren määrän eri josekeja, voi hyödyntää tätä puutetta Fuegoa vastaan pelatessa.

Varsinainen go-pelimoottori käyttää hyväkseen koko laudan Monte-Carlo-puuhakua ja sen toteutus löytyy GoUct-kirjastosta [5, luku 3]. GoUct-kirjasto sisältää myös toteutukset esimerkiksi lokaaleille Monte-Carlo-puuhauille. Go spesifinen Monte-Carlo-puuhaku on laajennettu SmartGamen Monte-Carlo-puuhausta ja koko laudan Monte-Carlo-puuhaku on taas laajennettu go-spesifisestä Monte-Carlo-puuhausta [5, luku 3.1]. Siirtopolitiikka Fuegossa muistuttaa hieman MoGon vastaavaa. Syöntisiirrot, atarin puolustussiirrot tai siirrot, jotka vastaavat käsin poimittuja 3x3-muotoja, valitaan sikäli kun siirto tulee edellisen siirron viereen. Fuego-spesifisiin parannuksiin kuuluu siirtojen generointi kaksi vapautta omaaville ryhmille. Jos mitään siirtoa ei ole vielä valittu, siirroksi valitaan globaali syöntisiirto. Jos globaalia syöntisiirtoa ei ole, valitaan siirto satunnaisesti kaikkien laillisten siirtojen joukosta. Korvavuuspolitiikka yrittää siirtää taktisesti huonot siirrot viereiseen pisteeseen. Siirtoa ei koskaan valita, jos kaikki viereiset pisteet on jo täytetty saman värisellä kivellä, ellei jokin näistä kivistä ole atarissa. Passaus generoidaan vain jos mitään muuta siirtoa ei saatu tuotettua. Tämän jälkeen pohjatietoluokka alustaa puu-

hun siirrot, jotka siirtopolitiikka olisi pelannut, ja alustaa nämä positiivisilla arvoilla. Ylimääräistä bonusta annetaan esimerkiksi, jos siirto johtaa vastustajan kiven tai ryhmän atarointiin tai jos siirto on tarpeeksi lähellä jotain toista omaa kiveä. Vastaavasti omien kivien ataroinnista sakoitetaan. Pelin lopun voitto/häviö-arviointia on muokattu hieman antamalla pieni bonus siirroille, jotka johtavat lyhyempiin määrään siirtoja pelissä ja lopputuloksena on kuitenkin suurempi pistemäärä. Tämän johdosta siirrot tuntuvat enemmän ihmismäisiltä, mutta tämä vaikuttaa myös positiivisesti suoraan pelivahvuuteen.

Go-pelimoottorin pelaajaluokka käyttää koko laudan Monte-Carlo-puuhakua generoimaan siirtoja ja tarjoaa lisäksi ylimääräistä funktionaalisuutta [5, luku 3.2]. Pelaaja voi määritellä esimerkiksi hakuparametreja riippuen nykyisestä laudan koosta. Jos pohdinta on kytketty päälle, pelaaja suorittaa haun nykyiselle laudan tilanteelle vastustajan miettiessä omaa siirtoaan. Haku keskeytetään, jos juurisolmu on lähellä voittoa ja ylimääräinen haku ajetaan tarkistamaan, onko lopputuloksena yhä voitto, jos siirto passataan.

Fuego vaikuttaa hyvin lupaavalta ohjelmistolta ja on myös pärjännyt hyvin aikaisemmissa turnauksissa muita ohjelmia vastaan [5, luku 3.4.2]. Lisäksi Fuego pääsi vuoden 2009 turnauksessa jo toiselle sijalle [11]. Fuegon säätövaran takia se pelaa myös varsin hyvin kotikoneillakin ja aloittelevat pelaajat saavatkin Fuegosta hyvän vastuksen. Kuvassa 6.1 näkyy Fuego pelaamassa itseään vastaan GoGui-käyttöliittymän päällä. Kuvassa näkyy myös pääte, josta voi syöttää käsin GTP:n mukaisia komentoja Fuegolle tai vaihtoehtoisesti valita näitä komentoja analyze-ikkunasta.



Kuva 6.1: Fuego pelaa itseään vastaan. Näkyvillä myös eri säätöjä Fuegolle.

7 Yhteenveto

Menetelmät ovat kehittyyneet viime vuosina melko paljonkin ja varsinkin Monte-Carlo-puuhakujen yleistyessä myös ohjelmistojen vahvuus on kasvanut reilusti. Ohjelmistot eivät kuitenkaan ole vielä täydellisiä. Miljoonat ihmiset, mukaanlukien nuoret lapset, voivat vielä pelata Gota reilusti paremmin kuin tietokone. Vaikka Go on onnistuttu jo ratkaisemaan 5x5-laudalle ja sitä pienemmillä, eivät samat menetelmät enää toimi suoraan 6x6- ja suurempien lautojen täydelliseen ratkaisemiseen [5]. Neuroverkkojen ja Monte-Carlon hyödyntäminen vaikuttavat hyvin lupaavilta ja onkin mielenkiintoista nähdä näiden kehittyminen tulevaisuudessa yrittäessä ratkaista Gota. Edellä mainitut menetelmät eivät vielä toimi kokonaisella 19x19-laudalla ja testeissä onkin käytetty lähinnä vain 9x9-lautoja. GNU Go:n suurimpina ongelmina tuntuvat olevan globaalin tilanteen hahmottaminen, strateginen peluu ja kriittisten pisteiden erottaminen suurista. GNU Go laskee melko hyvin lokaalit tilanteet laudalla, mutta ”unohtaa” usein koko laudan tilanteen. Kriittisillä pisteillä tarkoitetaan tässä, ettei GNU Go läheskään aina tiedä tarkalleen, milloin jotain ryhmää pitäisi suojata pelaten monesti suuria pistesiirtoja kriittisten suojaussiirtojen sijaan, jolloin ryhmän status yleensä romahtaa tai kuolee kokonaan pois. Fuego sen sijaan käyttää vahvasti hyödykseen Monte-Carlo-puuhakuja ja onkin jo jättänyt GNU Go:n ja muut tässä tutkielmassa esitellyt ohjelmistot taakseen. Monte-Carlo-puuhakuun perustuvien ohjelmistojen katsotaan myös olevan jo melko lähellä ratkaista 7x7-laudan peli [5, luku 1.1]. Menetelmä on muutenkin antanut melko vakuuttavia tuloksia eri turnauksissa, missä voittajat perustuvat aina tavalla tai toisella Monte-Carlo-puuhakuihin.

Lähteet

[1] American Go Association, *Computer Beats Pro at U.S. GO Congress*, saatavilla WWW-muodossa <URL: http://www.usgo.org/index.php?%23_id=4602>, viitattu 28.8.2009.

[2] B. Bouzy ja B. Helmstetter, *Monte-Carlo Go Developments*, saatavilla PDF-muodossa <URL: http://www.ai.univ-paris8.fr/~bh/articles/acg10-mcgo.pdf>, viitattu 26.8.2009.

[3] Jay Burmeister ja Janet Wiles, *The Challenge of Go as a Domain for AI Research: A Comparison Between Go and Chess*, saatavilla PDF-muodossa <URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.91.5615&rep=rep1&type=pdf>, viitattu 30.8.2009.

[4] Pierre-Arnaud Coquelin ja Rémi Munos, *Bandit Algorithms for Tree Search*, saatavilla PDF-muodossa <URL: http://arxiv.org/abs/cs/0703062>, viitattu 30.8.2009.

[5] Markus Enzenberger ja Martin Müller, *Fuego - An Open-source Framework for Board Games and Go Engine Based on Monte-Carlo Tree Search*, saatavilla PDF-muodossa <URL: http://www.cs.ualberta.ca/TechReports/2009/TR09-08/TR09-08.pdf>, viitattu 28.8.2009.

[6] Fuego-Project, *Fuego*, saatavilla WWW-muodossa <URL: http://fuego.sourceforge.net/>, viitattu 28.8.2009.

[7] Fuego-Project, *Fuego documentation*, saatavilla WWW-muodossa <URL: http://www.cs.ualberta.ca/~games/go/fuego/fuego-doc/>, viitattu 28.8.2009.

[8] GNU Go-Project, *GNU Go - GNU Project*, saatavilla WWW-muodossa <URL: http://www.gnu.org/software/gnugo/>, viitattu 26.8.2009.

[9] GNU Go-Project, *GNU Go Documentation*, saatavilla WWW-muodossa <URL: http://www.gnu.org/software/gnugo/gnugo_toc.html>, viitattu 26.8.2009.

[10] Alex Lubberts ja Risto Miikkulainen, *Co-Evolving a Go-Playing Neural Network*, saatavilla PDF-muodossa <URL: http://nn.cs.utexas.edu/downloads/papers/lubberts.coevolution-gecco01.pdf>, viitattu 26.8.2009.

[11] Martin Müller, *Fuego at the Computer Olympiad in Pamplona 2009: a Tournament Report*, saatavilla PDF-muodossa <URL: http://www.cs.ualberta.ca/TechReports/2009/TR09-09/TR09-09.pdf>, viitattu 4.9.2009.

[12] Erik C.D. van der Werf, H. Jaap van den Herik ja Jos W.H. Ui-terwijk, *Solving Go on Small Boards*, saatavilla PDF-muodossa <URL: http://erikvanderwerf.tengen.nl/pubdown/solving_go_on_small_boards.pdf>, viitattu 26.8.2009.