Othello-pelin evaluointifunktion kehittäminen

Samuli Siivonen

Helsinki 5. elokuuta 2003

Pro gradu -tutkielma

HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen laitos

Othello-pelin evaluointifunktion kehittäminen

Samuli Siivonen

Pro gradu -tutkielma
Tietojenkäsittelytieteen laitos
Helsingin yliopisto
5. elokuuta 2003, 63 sivua + 4 liitesivua

Tämän tutkielman aiheena ovat kahden pelaajan täyden informaation nollasummapelit. Erityisesti tarkastellaan menetelmiä, joilla tietokone oppii pelaamaan tällaisia pelejä. Esimerkkipeliksi on valittu Othello. Tutkielman osana on toteutettu Othelloa pelaava Java-ohjelma sekä opetusympäristö, jossa ohjelman pelitaso paranee. Tutkielmassa esitellään kahden pelaajan nollasummapelien pelipuut ja niiden läpikäyntialgoritmi minmax sekä läpikäynnin karsintamenetelmä alfa-beta. Oppimismenetelmistä keskitytään pelien evaluointifunktioiden automaattiseen parantamiseen. Tähän esitellään kaksi eri menetelmää: optimointi ja geneettinen ohjelmointi. Optimoinnissa evaluointifunktion parantamista käsitellään optimointiongelmana. Geneettisessä ohjelmoinnissa evaluointifunktioita parannetaan evoluution periaattein. Molempia menetelmiä on testattu käytännön testein ja tulokset on raportoitu tutkielmassa.

Aiheluokat (Computing Reviews 2002): I.2.6, I.2.8, G.1.6

Avainsanat: koneoppiminen, evaluointifunktiot, pelit, geneettinen ohjelmointi, optimointi, Othello

Sisältö

1 Johdanto

Pelit ovat olleet tekoälytutkimuksen kohteena jo aivan alan alkuajoista lähtien. Ne tarjoavat keinon kehittää ja testata erilaisia tekoälyn tekniikoita, kuten koneoppimis- ja hakumenetelmiä. Lisäksi pelien pelaamista voidaan pitää älykkäänä toimintana, joten pelejä osaavien koneiden kehittäminen on tietyssä mielessä tekoälyn kehittämistä.

Nykyään peliohjelmat ovat niin kehittyneitä, että tietokone on voittanut esimerkiksi Shakin hallitsevan maailmanmestarin [Sch97]. Monia pelejä, kuten Tammea ja Othelloa, tietokoneet pelaavat selkeästi paremmin kuin ihmiset. Osa peleistä, kuten Jätkänshakki (*Go-Moku*) ja Mylly (*Nine-Mens-Morris*), on ratkaistu [Her02, sivu 279]. Tämä tarkoittaa, että pelin lopputulos on selvä jo ennen kuin peli alkaa ja jokaiselle pelitilanteelle tiedetään paras mahdollinen siirto. Toisten pelien kohdalla kuitenkin, kuten Gon, tietokoneet saavuttavat kovista kehitysyrityksistä huolimatta vain keskinkertaisen pelitason [Her02, sivu 297].

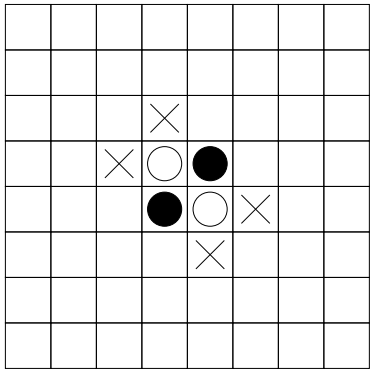
Tekoälyn kannalta kiinnostavia eivät ole niinkään ohjelmat, joiden toiminta perustuu vain mahdollisimman monen siirtomahdollisuuden läpikäyntiin. Pikemminkin kiinnostavia ovat ohjelmat, jotka esimerkiksi oppivat pelaamaan tai perustavat siirtonsa jonkinlaiseen intuitioon pelilaudan asemasta.

Tässä tutkielmassa käsitellään joitain menetelmiä, joiden avulla voidaan luoda pelejä pelaavia tietokoneohjelmia, ja pyritään opettamaan tällaisia ohjelmia pelaamaan paremmin. Menetelmien soveltamisen esimerkkipeliksi on valittu laajasti pelattu lautapeli Othello ja tutkielman osana on toteutettu Othelloa pelaava ohjelma. Ohjelman teon tarkoituksena on ollut toteuttaa käytännössä tässä tutkielmassa käsiteltyjä aiheita ja testata miten esiteltävät oppimismenetelmät onnistuvat parantamaan ohjelman pelaamista.

1.1 Othello

Othello on kahden pelaajan lautapeli, jossa molemmilla pelaajilla on pelikiekkoja, joiden toinen puoli on musta ja toinen valkoinen. Toinen pelaajista pelaa mustilla puolilla ja toinen valkoisilla. Pelilautana on 8×8 ruudukko ja pelin alkuasetelmassa laudan neljässä keskimmäisessä ruudussa on kaksi mustaa ja kaksi valkoista kiekkoa (kuva 1).

Peli etenee siten, että musta aloittaa. Pelaajat asettavat vuorotellen pelikiekkoja



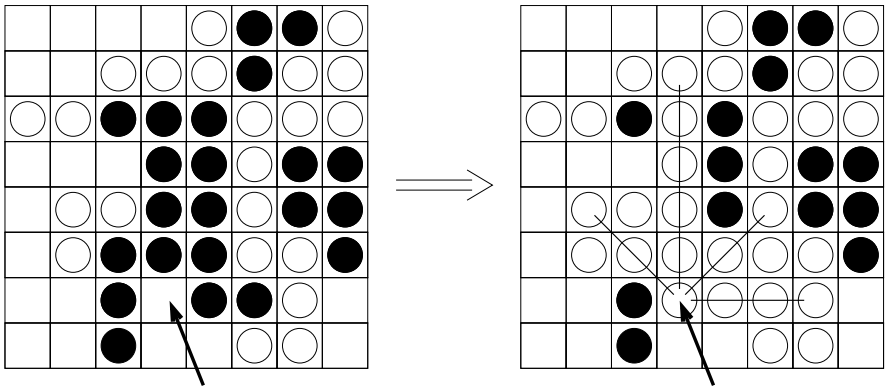
Kuva 1: *Othellon alkuasetelma ja kaikki mahdolliset mustan alkusiirrot (merkitty rasteilla).*

oma värinsä ylöspäin pelilaudan vapaille ruuduille. Sallittuja ruutuja ovat ruudut, jotka jättävät vastustajan kiekon tai kiekkoja asetetun kiekon ja toisen samanvärisen kiekon väliin (vaakasuoraan, pystysuoraan tai viistoon)(kuvat 1 ja 2).

Siirron jälkeen käännetään kaikki uuden kiekon ja jonkin toisen samanvärisen kiekon väliin jäävät vastustajan kiekot (kuva 2). Kääntämisen jälkeen niistä tulee siirron tehneen pelaajan kiekkoja. Pelaajat siirtävät vuorotellen aina kun se on mahdollista. Jos käy niin, että toisella pelaajista ei ole laillisia siirtoja, jää hänen siirtonsa väliin. Pelaajan pelikiekot eivät voi loppua ennen laudan täyttymistä. Peli loppuu, kun kummallakaan pelaajalla ei ole enää laillisia siirtoja. Lopputilanteessa lasketaan kuinka suuri osa pelikiekoista on valkoinen puoli ylöspäin ja kuinka suuri osa musta puoli ylöspäin. Jos mustia kiekkoja on enemmän kuin valkoisia, on musta pelaaja voittaja. Jos valkoisia kiekkoja on enemmän, on valkoinen pelaaja voittaja. Jos kiekkoja on yhtä paljon, on kyseessä tasapeli. [Bur97b, sivu 5]

1.2 Tutkielman esimerkkisovellus

Tutkielman osana on tehty sovellus, jossa on käytännössä toteutettu tutkielmassa esitellyt aiheet. Sovellus on Othelloa pelaava ohjelma sekä kehitysympäristö sen pelitason parantamiseksi. Sovellus on saanut nimen **ImpOthello** (**Imp**roving **Othello**) ja se on toteutettu Java-ohjelmointikielellä. Koko sovelluksen arkkitehtuurin kuvaus on liitteessä 1 ja tekninen dokumentointi sekä lähdekoodit löytyvät



Kuva 2: *Esimerkki Othellon siirrosta. Valkoinen asettaa kiekon nuolella osoitettuun kohtaan. Oikeanpuoleiseen kuvaan on merkitty käännetyt kiekot viivalla.*

sovelluksen WWW-sivulta [ImpOthello].

Sovelluksen näkyvä osa on Othelloa pelaava ohjelma. Käyttöliittymästä on kaksi erillistä versiota. Toinen versio on itsenäinen sovellus, joka aukeaa omaan ikkunaansa ja toinen on Internet-sivulla toimiva sovelma (applet), joka on toiminnaltaan melkein samanlainen kuin itsenäinen sovellus. Käyttöliittymän käyttöohjeet on kuvattu liitteessä 2. Käyttöliittymän sovelmaa voi kokeilla ImpOthellon WWW-sivulla [ImpOthello].

Esimerkkisovelluksen ulospäin näkyvä osa on Othelloa pelaava ohjelma. Itse tutkielman aihe on kuitenkin se miten ohjelma päättää tekemänsä siirrot. Siirtojen eteenpäin laskemista käsitellään tarkemmin luvussa 2. Luvussa 3 esitellään puolestaan miten tietokone pystyy arvioimaan pelitilanteita ja luvuissa 4 ja 5 käsitellään sitä miten tietokone oppii arvioimaan pelitilanteita paremmin. Kunkin luvun asiat on toteutettu käytännössä ImpOthellossa ja nämä käytännön toteutukset on kuvattu pääpiirteissään vastaavissa luvuissa.

2 Pelipuut ja niiden läpikäynti

Tässä luvussa esitellään pelipuun käsite ja sen läpikäynti yleisessä kahden pelaajan täyden informaation nollasummapelin tapauksessa. Läpikäyntiin liittyen esitellään minmax-algoritmi, joka suorittaa pelipuun läpikäyntiä pelin pelaajien toimia simuloiden, ja alfa-beta -karsinta, joka vähentää minmax-algoritmin läpikäymien solmujen määrää. Luvun lopussa kuvataan vielä miten esiteltävät asiat on toteutettu tutkielmaan liittyvässä käytännön toteutuksessa.

2.1 Kahden pelaajan täyden informaation nollasummapelit

Peliteoria tutkii erilaisten pelien pelaamista [Kiy87, sivu 662]. Se on saanut alkunsa perinteisten lautapelien, kuten Tammi (*Checkers*), tutkimisesta. Nykyään peliteoria on laajentunut käsittämään hyvin monia eri elämän osa-alueita, kuten taloustiedettä [Kre98] ja lakitiedettä [Bai98]. Peliteorian alkuajoilta on lähtöisin pelien luokittelu eri tyyppeihin. Tämän luokittelun mukaan tutkielmassa käsiteltävät pelit ovat kahden pelaajan täyden informaation nollasummapelejä (*two-player zero-sum games with perfect information*).

Täyden informaation peleissä pelitilanteissa on tiedossa kaikki seuraavat mahdolliset siirrot ja näiden vaikutukset [Bur63, sivut 17-18]. Suurin osa lautapeleistä on täyden informaation pelejä, koska pelin säännöt määräävät kaikki mahdolliset siirrot ja näiden aiheuttamat vaikutukset pelin kulkuun. Nollasummapelit ovat sellaisia, joissa pelaajien pyrkimykset ovat päinvastaisia. Yhden pelaajan hyöty on samalla toisten pelaajien tappio ja pelaajien välinen yhteistyö ei edistä heidän tavoitteitaan [Bur63, sivu 20].

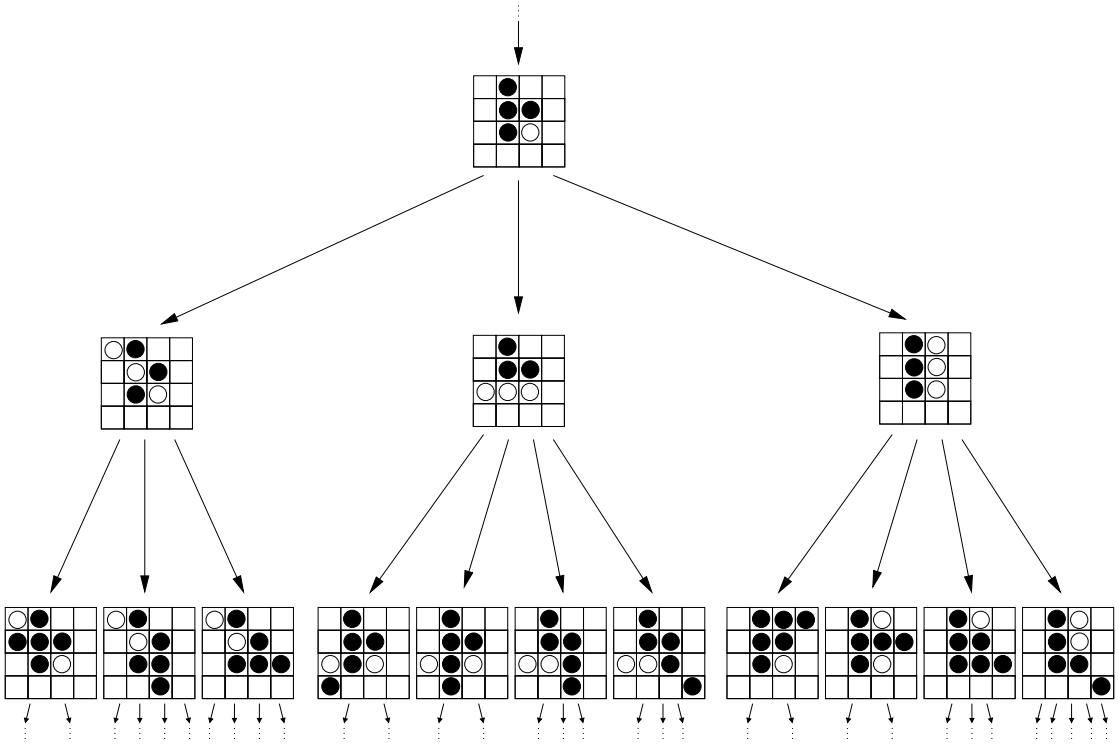
Suurin osa perinteisistä kahden pelaajan lautapeleistä, kuten Shakki, Tammi, Mylly ja Go, on kahden pelaajan täyden informaation nollasummapelejä. Näissä kaksi pelaajaa mittelee toisiaan vastaan tehden vuorotellen siirtoja tavoitellen vastakkaisia päämääriä.

2.2 Pelipuut

Usein peleissä on järkevää laskea eteenpäin mahdollisia siirtoja, jotta pystytään ennakoimaan tulevia pelitilanteita. Siirtojen eteenpäin laskemisessa voidaan käyttää pelipuita. Ne ovat puita, joissa solmuina on pelin tilanteita ja kunkin solmun

jälkeläisinä ovat ne pelitilanteet, joihin päästään emosolmusta jollain laillisella siirrolla [Nil98, sivu 199]. Pelipuun juurena on pelin alkutilanne, tämän jälkeläisinä ovat tilanteet joihin päästään alkutilanteesta, näiden jälkeläisinä ovat tilanteet, joihin päästään jälkeläissolmujen pelitilanteista ja niin edelleen. Pelipuun lehtinä ovat kaikki mahdolliset pelin lopputilanteet. Pelipuu sisältää pelin kaikki mahdolliset siirrot.

Kuvassa 3 on osa 4×4 -laudan Othellon pelipuusta. Pelin säännöt ovat samat kuin Othellon, mutta pelilauta on vain 4×4 ruutua. Näin pienellä laudalla Othelloa ei ole järkevä pelata, mutta kuva valaisee pelipuun ideaa. Joskus pelipuiksi kutsutaan myös pelipuun osia.



Kuva 3: *Osa* 4×4 *-laudan Othellon pelipuusta.*

Pelipuun koko kasvaa eksponentiaalisesti siirtojen lukumäärän suhteen. Jos pelissä on keskimäärin k kappaletta siirtoja pelitilannetta kohti ja peli kestää keskimäärin s siirtoa, on pelipuun koko luokkaa $O(k^s)$ [Nil98, sivu 207]. Eksponentiaalinen kasvu aiheuttaa sen, että kun peli monimutkaistuu tarpeeksi, ei koko pelipuun läpikäynti ole käytännössä mahdollista. Jos esimerkiksi Othellossa jokaisessa pelitilanteessa pelaajalla olisi viisi mahdollista siirtoa ja peli kestäisi kes-

kimäärin 50 siirtoa (lauta täyttyy 60 siirron jälkeen) saataisiin Othellon pelipuun koolle arvio $O(5^{50})$ solmua. Tämä arvio on alaraja pelipuun koolle, sillä Othellossa on yleensä 5-15 siirtomahdollisuutta [Rus95, sivu 138] ja pelit kestävät yleensä yli 50 siirtoa. Lähempänä oikeaa kokoa lienee arvio $O(10^{58})$ [Her02, sivu 300]. Kun lasketaan

$$\frac{5^{50}\,solmua}{(10^9\,solmua/s) \times (60s \times 60 \times 24 \times 365 \times 15 \times 10^9)} \approx 188 \times 10^6,$$

nähdään, että pelipuun läpikäymiseen menee miljardin solmun sekuntivauhdilla noin 188 miljoona kertaa maailmankaikkeuden ikä $(60s \times 60 \times 24 \times 365 \times 15 \times 10^9)$ [Oja00, sivu 135]. Miljardin solmun sekuntivauhtiin ei päästä millään ohjelmalla nykyisissä tietokoneissa, mutta on kehitetty erityistä laitteistoa, joka kykenee lähes tähän vauhtiin [Cam02, sivu 58].

Koska pelipuut ovat suuria, ei koko pelipuun läpikäynti ole mahdollista kuin kaikkein yksinkertaisimpien pelien tapauksissa. Yleensä tyydytäänkin tutkimaan pelipuiden osia ja kehitetään menetelmiä läpikäytävien solmujen määrän vähentämiselle.

2.3 Minmax-algoritmi

Kutsutaan kahden pelaajan nollasummapelin pelaajia nimillä *min* ja *max*. Olkoon *max* pelin aloittaja. Koska pelaajien tavoitteet ovat päinvastaiset, ovat pelaajan *max* parhaat siirrot periaatteessa huonoimpia pelaajalle *min* ja päinvastoin. Tämä edellyttää tietysti sitä, että analysoimme siirtoja oikein. Siirron analysoinnissa käytetään apuna evaluointifunktioita. Niitä käsitellään tarkemmin luvussa 3. Tässä vaiheessa riittää tietää, että evaluointifunktio on kuvaus, joka liittää jokaiseen pelitilanteeseen jonkin reaaliluvun. Tämä luku kuvaa pelitilanteen suotuisuutta pelaajille. Määrätään, että mitä pienempi evaluointifunktion arvo on, sitä parempi pelitilanne on pelaajalle *min* ja mitä suurempi arvo on, sitä suotuisampi tilanne on pelaajalle *max*.

Minmax-algoritmin periaatteena on käydä läpi pelipuuta olettaen, että pelaaja tekee aina optimaalisen siirron evaluointifunktioiden osoittaman arvon mukaan [Hyv93, sivu 180]. Täten siis alussa pelaaja *max* valitsee mahdollisista siirroista sen, jota vastaavan pelitilanteen evaluointifunktion arvo on suurin. Tämän jälkeen *min* tekee siirron, jonka evaluointifunktion arvo on pienin ja niin edelleen. Itse algoritmissa käydään läpi koko pelipuu, mutta käytännössä joudutaan usein

rajoittumaan pelipuun osaan [Rus95, sivu 124]. Tämän jälkeen palataan tätä puuta takaisinpäin ja määrätään puun solmujen (eli pelitilanteiden) evaluointien arvoja seuraavalla tavalla [Rus95, sivu 124]:

- Jos solmulla ei ole lapsia evaluoinnin arvoksi tulee evaluointifunktion arvo.
- Jos solmulla on lapsi(a) ja pelivuorossa on *max*, evaluoinnin arvoksi tulee *max* { lasten evaluointien arvot }.
- Jos solmulla on lapsi(a) ja pelivuorossa on *min*, evaluoinnin arvoksi tulee *min* { lasten evaluointien arvot }.

Puun läpikäynti täytyy toteuttaa jälkijärjestyksenä (*postorder*) eli siten, että solmun jälkeläiset käydään läpi ennen emosolmua [Wik02, luku 3.1.5].

Minmax-algoritmin ideana on saada parempi arvio pelitilanteille laskemalla siirtoja eteenpäin. Algoritmin suoritus ei ole sidottu alkamaan pelin alkutilanteesta vaan mitä tahansa tilannetta voidaan pitää algoritmin alkutilanteena ja vuorossa voi olla kumpi tahansa pelaaja.

Oletetaan, että evaluointifunktio antaa voittotilanteessa arvon ∞ tai $-\infty$ sen mukaan kumpi on voittanut pelin. Tällöin pelaaja huomaa varman voiton tai häviön minmax-algoritmin avulla niin monta siirtoa aiemmin, kun algoritmi laskee siirtoja eteenpäin [Nil98, sivu 199]. Jos algoritmi laskee kaikki mahdolliset siirrot pelitilanteesta eteenpäin, saadaan tieto siitä, kumman voittoon pelitilanne päättyy molempien pelatessa optimaalisesti [Rus95, sivu 124].

2.4 Alfa-beta -karsinta

Alfa-beta -karsinta on menetelmä, jolla karsitaan minmax-algoritmin tekemää turhaa työtä [Rus95, sivut 129-131]. Minmax-algoritmi käy läpi kaikki pelipuun solmut. Oletetaan tilanne, jossa algoritmin suoritus saapuu solmuun, jolla on viisi jälkeläistä. Näistä siirroista yksi on suora voitto vuorossa olevalle pelaajalle. Kun solmun jälkeläisiä käydään läpi ja tullaan voittosiirtoa vastaavaan jälkeläisen kohdalle, ei solmun evaluointi voi enää muuttua. Täten jälkeläisten läpikäynti voitaisiin lopettaa, mutta minmax-algoritmi käy joka tapauksessa läpi kaikki solmun jälkeläiset.

Edellä esitetyn esimerkin kaltaista tilannetta voidaan laajentaa koskemaan muitakin kuin vain varmoja voittotilanteita. Määritellään kaikille *max*-solmuille α-arvo ja kaikille *min*-solmuille β-arvo seuraavalla tavalla [Nil98, sivu 204]:

- *Max*-solmun α-arvo on suurin sen jälkeläisen palauttama evaluoinnin arvo.
- *Min*-solmun β-arvo on pienin sen jälkeläisen palauttama evaluoinnin arvo.

Täten α- ja β-arvot ovat solmujen evaluointien arvoja minmax-algoritmin edetessä kunkin jälkeläisen läpikäynnin jälkeen. Määritelmistä nähdään, että α-arvo voi vain kasvaa ja β-arvo vain vähetä. Täten α-arvo on alaraja *max*-solmun evaluoinnille ja β-arvo yläraja *min*-solmun evaluoinnille.

Jos algoritmin edetessä solmun ja sen emon α- ja β-arvoille pätee $\alpha \geq \beta$, voidaan algoritmin suoritus katkaista tämän solmun osalta ja siirtyä eteenpäin. Tämä pätee koska [Nil98, sivut 203-204]:

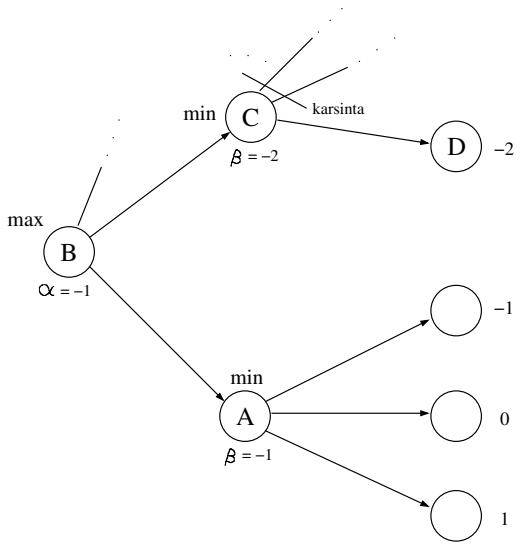
- Jos tarkastelussa oleva solmu S on *min*-solmu, niin solmun β-arvo on pienempi kuin sen emon α-arvo. $\Leftrightarrow$ Solmun S evaluoinnin yläraja on pienempi kuin sen emon jo saavuttama evaluoinnin arvo.

  Koska emo on *max*-solmu, se pyrkii mahdollisimman suureen evaluoinnin arvoon. $\Rightarrow$ Solmun S evaluointi ei voi koskaan saada sellaista arvoa, joka vaikuttaisi sen emon evaluoinnin arvoon.

- Jos tarkastelussa oleva solmu S on *max*-solmu, niin solmun α-arvo on suurempi kuin sen emon β-arvo. $\Leftrightarrow$ Solmun S evaluoinnin alaraja on suurempi kuin sen emon jo saavuttama evaluoinnin arvo.

  Koska emo on *min*-solmu, se pyrkii mahdollisimman pieneen evaluoinnin arvoon. $\Rightarrow$ Solmun S evaluointi ei voi koskaan saada sellaista arvoa, joka vaikuttaisi sen emon evaluoinnin arvoon.

Kun solmun läpikäynti keskeytetään, voidaan sen evaluoinnin arvoksi määrätä sen hetkinen ylä- tai alaraja eli *max*-solmuille evaluoinnin arvoksi tulee solmun α-arvo ja *min*-solmuille solmun β-arvo. Tämä arvo ei välttämättä ole sama kuin täydellisen minmax -algoritmin suorituksen jälkeen. Arvo ei kuitenkaan vaikuta siihen, mikä kussakin pelitilanteessa on pelaajan paras siirto [Nil98, sivu 205].



Kuva 4: *Esimerkki alfa-beta karsinnasta.*

Kuvassa 4 on esimerkki alfa-beta -karsinnasta. Tarkastellaan kuvan tilannetta solmussa B, kun minmax-algoritmi on jo käynyt solmussa A ja on seuraavaksi siirtymässä solmuun C. Tällöin solmun A β-arvo on -1, koska sen jälkeläisten palauttamista arvoista pienin on -1. Se on samalla solmun evaluointiarvo. Solmun B α-arvo on tässä vaiheessa -1, koska solmu A on palauttanut arvon -1. Siirrytään nyt solmuun C. Solmun C ensimmäinen jälkeläinen, solmu D, palauttaa arvon -2. Nyt solmun C β-arvoksi tulee -2, koska sen jälkeläinen on palauttanut tämän arvon. Tässä vaiheessa huomataan, että solmun C läpikäynti voidaan keskeyttää, koska solmun C β-arvo on pienempi kuin sen emon α-arvo. Tällöin siis ehto $\alpha \geq \beta$ täyttyy. Solmun C evaluoinnin arvoksi asetetaan β eli -2 ja sen muita lapsia ei läpikäydä vaan palataan solmuun B ja ryhdytään käymään läpi tämän muita jälkeläisiä.

Karsinta johtuu siitä, että kun solmun C β-arvoksi tulee -2, tiedetään, että solmun C pelitilanteessa pelaaja *min* voi tehdä siirron, joka johtaa evaluointiin -2. Toisaalta koska solmun B α-arvo on -1, tiedetään, että pelaaja *max* voi tehdä solmun B pelitilanteessa siirron, joka johtaa evaluointiin -1. Koska pelaaja *max* yrittää maksimoida evaluoinnin arvoa, ei solmun C pelitilanteen valitseminen voi mitenkään johtaa parempaan tulokseen kuin solmun A pelitilanteen valitseminen. Tästä syystä pelaajan *max* ei tarvitse käydä läpi muita solmun C jälkeläisiä.

Edellä mainittiin, että alfa-beta -karsinta ei vaikuta mitenkään optimaalisen siirron valintaan. Tämä pitää paikkansa vain jos siirtoja valittaessa valitaan aina järjestyksessä ensimmäinen parhaista siirroista ja järjestys on sama kuin karsinta-

algoritmin etenemisjärjestys. Muussa tapauksessa saattaa käydä niin, että pelaaja ei siirrä parasta siirtoa.

Jos kuvassa 4 solmun D evaluointi olisi -1, olisi solmun C β-arvo -1 ja karsinta tapahtuisi. Tällöin solmun C evaluoinniksi tulisi -1. Nyt solmujen A ja C evaluoinnit olisivat yhtä hyvät ja niitä vastaavat siirrot olisivat periaatteessa yhtä hyvät. Solmun C oikea evaluointi voisi kuitenkin olla vaikka -∞, jos jonkin sen jälkeläisen evaluointi olisi -∞. Tässä tapauksessa siirron C valitseminen johtaisi pelaajan *max* häviöön vaikka siirrot A ja C vaikuttavat yhtä hyviltä.

Jos halutaan, että yhtä hyvien siirtojen kohdalla arvotaan tehtävä siirto, on karsintaehdoksi muutettava $\alpha > \beta$. Tällöin karsitaan solmut, jotka tiedetään varmasti huonommiksi (ei yhtä huonoiksi) kuin jokin tiedossa oleva solmu ja täten niiden evaluointien arvoiksi tulee varmasti huonompi kuin paras tähän mennessä löydetty evaluointi.

Alfa-beta -karsinta on periaatteessa tehokkaampi kuin pelkkä minmax -algoritmi. Kuitenkin solmujen läpikäyntijärjestyksellä on suuri merkitys karsinnan tehokkuuteen [Rus95, sivu 131]. Olkoon pelipuun keskimääräinen haarautumisaste k. Täten siis puun solmuilla on keskimäärin k lasta. Jos solmun jälkeläiset on järjestetty optimaalisesti, saadaan alfa-beta -karsinnalla puu, jonka keskimääräinen haarautumisaste on $\sqrt{k}$ [Nil98, sivu 207]. Jos taas järjestys on mahdollisimman huono, ei karsintaa tapahdu lainkaan. Tällöin käytännössä alfa-beta -karsinta saattaa olla monimutkaisempana toteutuksena hieman hitaampi kuin minmax -algoritmi.

Tutkimuksien mukaan satunnaisen järjestyksen tapauksessa alfa-beta -karsinnalla saadaan puun keskimääräiseksi haarautumisasteeksi $\sqrt[4]{k^3}$ [Nil98, sivu 207], joten yleensä karsinnasta on hyötyä. Lisäksi hyvällä järjestämisellä päästään lähelle optimaalista karsintaa, joka nopeuttaa jo huomattavasti algoritmin läpikäyntiä. Optimaalisen järjestyksen tietäminen edellyttäisi oikeiden evaluointien tietämistä ja täten se ei ole mahdollista tai järkevää, mutta esimerkiksi järjestämällä *min*-solmujen jälkeläiset evaluointifunktion mukaisten evaluointien mukaan nousevaan järjestykseen ja *max*-solmujen jälkeläiset laskevaan järjestykseen, saadaan aikaan varsin hyvä karsinta [Nil98, sivu 207].

2.5 Pelipuun toteutus

Tutkielman sovelluksessa, ImpOthellossa, pelipuut on toteutettu omana luokkanaan `GameTreeNode` (katso liite 1). Luokan ilmentymät ovat pelipuun solmuja. Jokaisessa solmussa on pelilaudan tilannetta kuvaava matriisi, jossa 1 merkitsee mustaa kiekkoa, -1 valkoista kiekkoa ja 0 tyhjää ruutua. Lisäksi jokaisessa solmussa ovat kentät, jotka ilmoittavat kumpi pelaajista on vuorossa, mikä on pelilaudan tilanteen evaluoinnin arvo ja solmun α- tai β-arvo. Jokaisessa solmussa on määritetty lisäksi solmun jälkeläiset eli pelipuun solmut, joiden pelitilanteisiin päästään yhdellä laillisella siirrolla alkuperäisestä tilanteesta. Pelipuun läpikäynti tapahtuu mistä tahansa solmusta alkaen käymällä läpi tämän solmun jälkeläiset, niiden jälkeläiset ja niin edelleen.

Tässä luvussa käsitellyt pelipuun läpikäyntialgoritmit on toteutettu Java-luokassa `Search` (katso liite 1). Tämä luokka sisältää metodeja, jotka käyvät läpi pelipuuta. Se sisältää rekursiiviset metodit, jotka toteuttavat puhtaan minmax-algoritmin ja alfa-beta -karsintaa käyttävän minmax-algoritmin. Lisäksi luokka sisältää metodin joka määrittää tietokoneen tekemän siirron. Metodi ajaa minmax-algoritmin parametrina saadusta pelipuun solmusta alaspäin ja palauttaa algoritmin suorituksen seurauksena saadun parhaan siirron. Jos parhaita siirtoja on useampia, arpoo kone niistä jonkun. Jos taas pelitilanne on varma häviö, tekee kone siirron, joka on evaluointifunktion mukaan paras.

Tarkemmat tiedot ImpOthellon toteutuksesta löytyvät sen kotisivuilta [IMPOTHELLO].

3 Evaluointifunktiot

Tässä luvussa esitellään evaluointifunktion käsite yleisellä tasolla. Tämän jälkeen tarkastellaan evaluointifunktion parantamista ja luvun lopussa on vielä esitelty miten tutkielman esimerkkisovelluksessa evaluointifunktiot ja niiden vertailu on toteutettu.

3.1 Evaluointifunktion käsite

Evaluointifunktioiksi (*evaluate/evaluation function*) kutsutaan hakupuissa erilaisia funktioita, jotka saavat syötteenään puun solmun ja palauttavat jonkin arvon (reaaliluvun) [Nil98, sivut 139-141]. Peleissä evaluointifunktiolla tarkoitetaan funktiota, joka saa syötteenään pelitilanteen ja palauttaa pelitilannetta kuvaavan reaaliluvun [Rus95, sivu 127].

Kahden pelaajan nollasummapeleissä reaaliluku yleensä ilmoittaa kummalle pelaajalle pelitilanne on edullisempi ja kuinka paljon. Tämä tapahtuu esimerkiksi määräämällä toiselle pelaajista positiiviset evaluoinnin arvot edullisiksi ja toiselle negatiiviset. Evaluoinnin itseisarvo ilmoittaa kuinka hyvä evaluointi on pelaajalle. Evaluointi 0 tarkoittaa pelitilanteen olevan yhtä hyvä molemmille pelaajille. Tässä tutkielmassa evaluointifunktion arvot on määritelty edellä kuvatulla tavalla. Samalla tavalla ne on määritelty myös luvussa 2.3 esitetyssä minmax-algoritmissa.

Pelien evaluointifunktiot voivat olla keskenään hyvin erilaisia. Ne voivat olla funktiota, jotka ovat muotoa $f : \mathbb{R}^n \rightarrow \mathbb{R}$, $n \in \mathbb{N}$. Pelilaudan ruudut voidaan koodata funktion parametreiksi ja kukin parametri ilmoittaa jollain kokonaisluvulla mikä pelinappula sitä vastaavassa ruudussa on. Evaluointifunktio suorittaa tämän jälkeen laskutoimituksia parametreille ja palauttaa evaluoinnin arvon. Esimerkiksi Logistello, erittäin vahva Othelloa pelaava ohjelma, käyttää tämän kaltaista lineaarista evaluointifunktiota [Bur97c, sivut 4-6]. Myös tässä tutkielmassa käsiteltävät evaluointifunktiot ovat edellä esitetyn kaltaisia.

Evaluointifunktio voi olla myös neuroverkko (*neural network*). Moriarty ja Miikkulainen käyttivät tutkimuksessaan [Mor93] Othellon evaluointifunktiona neuroverkkoa, jossa oli 128 erilaista syötettä (kaksi jokaista pelilaudan ruutua kohti) ja 64 tulostetta. Syöteparit ilmoittivat oliko ruudussa musta tai valkoinen kiekko ja tulosteet ilmoittivat kuinka suositeltava siirto vastaavaan ruutuun on.

3.2 Evaluointifunktion parantaminen

Tietokone pelaa sitä paremmin, mitä parempi sen käyttämä evaluointifunktio on. Tästä syystä evaluointifunktion parantaminen parantaa tietokoneen pelitasoa ja voidaan ajatella tietokoneen oppivan pelaamaan. Jos evaluointifunktion parantaminen tehdään automaattiseksi, tietokone oppii itsekseen pelaamaan. Erityisesti jos evaluointifunktion parantaminen perustuu tietokoneen itse pelaamiin peleihin, tietokone oppii paremmaksi pelaajaksi pelikokemuksen kautta.

Seuraavissa luvuissa esitellään kaksi tapaa parantaa evaluointifunktioita automaattisesti. Ne ovat yksinkertaisia ja toimivat mille tahansa pelille, mutta eivät tehokkuudeltaan ole parhaita mahdollisia. Molemmissa menetelmissä tietokone pelailee itsekseen pelejä ja kehittää evaluointifunktiotaan pelikokemuksensa perusteella.

3.3 Eräs testausympäristö evaluointifunktioille

Tutkielmassa esiteltävät evaluointifunktion parannusmenetelmät perustuvat siihen, että tiedetään evaluointifunktion hyvyys ja koetetaan saada aikaan parempi evaluointifunktio. Tämä evaluointifunktioiden paremmuuden mittaaminen on toteutettu yksinkertaisesti testaamalla miten evaluointifunktiot pärjäävät pelatessaan vastakkain. Tätä varten on toteutettu automaattinen testausympäristö, joka peluuttaa evaluointifunktioita keskenään ja määrittelee niiden paremmuusjärjestyksen. Tämä testausympäristö ja sen toteutus esitellään teoreettisella tasolla seuraavaksi. Tarkempi käytännön toteutus on kuvattu tutkielman sovelluksen WWW-sivulla [ImpOthello].

3.3.1 Evaluointifunktion toteutus

Toteutetussa sovelluksessa käytetään Othellon evaluointifunktiota, joka saa parametrinaan laudan tilanteen kokonaislukutaulukkona ja laskee evaluoinnin arvon kokonaisluvuista käyttäen ennalta määrättyjä binäärisiä operaatioita ja reaalilukuvakiota. Laudan tilanne on koodattu siten, että kukin pelilaudan ruutu vastaa yhtä parametrina saatavaa kokonaislukua -1, 0 tai 1. Nämä luvut vastaavat ruudun tiloja valkoinen kiekko, tyhjä ruutu ja musta kiekko. Binääriset operaatiot, jotka on valittu tutkielmassa käytettäväksi, ovat yksinkertaisimmat aritmeettiset operaatiot: yhteen-, vähennys-, kerto-, ja jakolasku ($+, -, \times, /$). Evaluointifunktion

palauttama arvo ilmoittaa pelitilanteen olevan hyvä mustalle, jos se on positiivinen ja hyvä valkoiselle, jos se on negatiivinen.

Evaluointifunktiot ovat muotoa $f_1 o_1 f_2 o_2 \dots f_{n-1} o_{n-1} f_n$, $n \in \mathbb{N}$, missä f_i, $i = 1, \dots, n$, on joko kaarisuluissa oleva evaluointifunktio, vakio tai muuttuja, ja o_i, $i = 1, \dots, n-1$, on jokin sallittu operaattori. Vakiot ovat Java-ohjelmointikielen reaalilukuja (`double`), ja muuttujat ovat muotoa $[i, j]$, missä $i, j \in \{0, \dots, 7\}$. Muuttujan $[i, j]$ arvoksi tulee evaluoinnissa -1, 0 tai 1 sen mukaan mikä on pelitilanteessa pelilaudan ruudun (i, j) tilanne. Ruutu $(0, 0)$ on pelilaudan oikeassa yläkulmassa, i kasvaa vasemmalta oikealle ja j ylhäältä alas. Muuttujia on siis yhteensä 64.

Sallitut operaattorit määritellään omassa luokassaan, jossa määritellään myös operaattorien suoritusjärjestys ja lähdekoodi suoritettavalle binääriselle operaatiolle. Vakiot ja muuttujat ovat oman luokkansa edustajia ja funktiot omansa. Funktioluokassa on metodi funktion evaluoinnin arvon laskemiselle. Evaluoinnin arvo lasketaan siten, että ensin lasketaan kunkin alifunktion evaluoinnin arvo ja muuttujien evaluointien arvot. Täten saadaan aikaan lauseke, jossa on lukuja ja niiden välissä operaattoreita. Lausekkeen arvo lasketaan operaattorien arvojärjestyksen mukaisesti vasemmalta oikealle. Tutkielmassa käytetyissä evaluointifunktioissa operaatioiden laskujärjestys on normaali matemaattinen laskujärjestys eli ensin lasketaan $\times$ ja / vasemmalta oikealle ja sitten + ja $-$ samoin.

3.3.2 Testausympäristön toteutus

Testausympäristön perusyksikkö on peli. Se on toteutettu omana Java-luokkanaan. Luokan edustaja luodaan antamalla pelissä pelaavien pelaajien evaluointifunktiot. Tämän jälkeen kutsutaan metodia, joka simuloi Othello-peliä, jossa pelaajat tekevät siirtonsa evaluointifunktioiden mukaan.

Yksi peli ei vielä kerro kumpi pelaajien käyttämistä evaluointifunktioista on parempi. Pelin lopputulos ei välttämättä ole riippumaton siitä kumpi pelin aloittaa ja peli ei välttämättä etene aina samalla tavalla. Kun pelitilanteiden evaluointien arvot ovat yhtä hyvät, joko arvotaan parhaiden siirtovaihtoehtojen väliltä pelattava siirto tai pelataan ensin löydetty paras siirto. Yhteen peliin voi vaikuttaa sattuma sen verran, että paremmalla evaluointifunktiolla pelaava pelaaja ei välttämättä voita peliä.

Kahden evaluointifunktion vertailuun on kehitetty oma luokkansa. Vertailuluokkaa käytetään määrittelemällä ensin vertailtavat evaluointifunktiot ja kutsumal-

la tämän jälkeen vertailumetodia. Metodi suorittaa kahden evaluointifunktion vertailun peluuttamalla evaluointifunktioita vastakkain asetuksien mukaan ja tekee raportin vertailusta. Asetuksissa voi määrätä vertailussa pelattavien pelien määrän ja vaihtuuko aloittava pelaaja. Evaluointifunktioiden vertailua käytetään apuna luvussa 4 esiteltävässä evaluointifunktion parannusmenetelmässä.

Luvussa 5 esiteltävässä geneettisessä ohjelmoinnissa ei riitä pelkästään kahden evaluointifunktion vertailu vaan siinä tarvitsee laittaa joukko evaluointifunktioita paremmuusjärjestykseen. Tätä varten on kehitetty oma luokkansa, joka simuloi evaluointifunktioiden turnausta. Se käyttää edellä mainittua vertailuluokkaa ja vertailee evaluointifunktioita joko toisiinsa tai asetuksissa määrättyyn vertailufunktioon. Vertailujen tulosten mukaan luokka laittaa evaluointifunktiot paremmuusjärjestykseen ja tekee raportin vertailuista ja tuloksista.

4 Evaluointifunktion parantaminen optimoimalla

Tässä luvussa esitellään, miten evaluointifunktiota voi parantaa optimoimalla. Tavoitteena on hakea mahdollisimman hyviä arvoja tietyn tyyppisten evaluointifunktioiden vakioiden arvoiksi. Ideana on kiinnittää evaluointifunktion tyyppi ja tämän jälkeen pyrkiä hiomaan tämän tyyppisen evaluointifunktion vakioiden arvoja sellaisiksi, että evaluointifunktio pärjäisi mahdollisimman hyvin. Aluksi muotoillaan ongelma formaalisti optimointiongelmana. Tämän jälkeen keksitään ratkaisuja tähän ongelmaan ja katsotaan miten ratkaisu on käytännössä toteutettu tässä tutkielmassa. Lopuksi esitellään tehdyt testit ja niiden tulokset.

4.1 Evaluointifunktion parantaminen optimointiongelmana

Hyvin monet käytännön ongelmat voidaan palauttaa optimointiongelmiksi. Optimointiongelman yleinen muoto voidaan esittää seuraavasti [Gil81, sivu 1]:

$$\begin{aligned} &\text{minimoi } F(x),\ x \in \mathbb{R}^n \\ &\text{ehdoilla } c_i(x) = 0,\ i = 1, 2, \ldots, m' \\ &\qquad\qquad c_j(x) \geq 0,\ j = m' + 1, \ldots, m. \end{aligned}$$

Funktion $F(x)$ maksimointiongelma voidaan esittää minimointiongelmana minimoimalla funktiota $-F(x)$. Kaikki yhtälöt ja epäyhtälöt voidaan muuttaa yllä esitettyyn muotoon siirtämällä kaikki termit yhtälön tai epäyhtälön toiselle puolelle ja tarvittaessa tarkastelemalla niiden negaatioita [Fle81, sivut 1-2].

Tavoitteena on esittää evaluointifunktion parantaminen optimointiongelmana. Tässä rajoitutaan tapaukseen, jossa on annettu jokin evaluointifunktio ja tavoitteena on löytää mahdollisimman hyvät arvot evaluointifunktion vakioille. Käsiteltävänä on evaluointifunktio $ef : T \to \mathbb{R}$, missä T on mahdollisten pelitilanteiden joukko. Jotta optimointiongelma saataisi mielekkääksi voidaan tehdä oletus, että evaluointifunktio ef sisältää joitain vakioita, jotka vaikuttavat evaluoinnin tulokseen. Tämä rajoite sulkee pois osan evaluointifunktiosta, mutta tälle osalle käsiteltävät optimointimenetelmät eivät ole mielekkäitä.

Olkoon $v_1, \ldots, v_n \in \mathbb{R}$, $n \in \mathbb{N}$ evaluointifunktion ef vakioita, jotka vaikuttavat evaluointiin. Määritellään ef_x, $x = (x_1, \ldots, x_n) \in \mathbb{R}^n$, siten että korvataan evaluointifunktion ef kukin vakio v_i reaaliluvulla x_i, $i = 1, \ldots, n$. Olkoon evaluointifunktion ef tyyppisten evaluointifunktioiden joukko $E_{ef} = \{ef_x | x \in \mathbb{R}^n\}$. Määritellään

nyt kuvaus $replace_{ef} : \mathbb{R}^n \to E_{ef}, replace_{ef}(x) = ef_x$. Määritellään lisäksi kuvaus $power_{ef} : E_{ef} \to \mathbb{R}$, joka liittää evaluointifunktioon ef_x reaaliluvun, joka kuvaa sitä kuinka hyvä evaluointifunktio on. Täten jos $power_{ef}(ef_{x_1}) > power_{ef}(ef_{x_2})$, on evaluointifunktio ef_{x_1} parempi kuin ef_{x_2}. Evaluointifunktion parantaminen voidaan nyt muuntaa optimointiongelmaksi:

$$\boxed{\text{minimoi } -power_{ef}(replace_{ef}(x)),\ x \in \mathbb{R}^n.}$$

Tässä etsitään vektoria $x = (x_1, \ldots, x_n)$. Kun sijoitetaan x:n skalaarit evaluointifunktion ef vakioiden paikalle, saadaan funktion $power_{ef}$ mukaan paras evaluointifunktio.

4.2 Optimointiongelman ratkaiseminen

Suurin osa tunnetuista optimointimenetelmistä perustuu tavalla tai toisella funktion gradientin, eli moniulotteisen derivaatan, laskemiseen tai muihin funktion säännöllisyyteen liittyviin menetelmiin [Gil81, sivu 93]. Tarkastellaan ensin tarkemmin optimoitavaa funktiota ja tämän jälkeen kehitetään joitain menetelmiä sen optimoimiseksi.

4.2.1 Optimoitava funktio

Olkoon t pelitilanne sellaisessa muodossa, että sen voi antaa evaluointifunktiolle ef parametrina. Määritellään nyt funktio

$$test_{ef(t)} : \mathbb{R}^n \to \mathbb{R}, test_{ef(t)}(x) = (replace_{ef}(x))(t).$$

Funktion $test_{ef(t)}$ arvo on pelitilanteen t evaluoinnin arvo evaluointifunktion ef eri vakioiden arvoilla. Tarkastelemalla funktiota eri pelitilanteilla t saadaan kuva siitä miten evaluointifunktion ef vakioiden arvojen muuttaminen vaikuttaa sen evaluointien arvoihin. Jos funktion $test_{ef(t)}$ kuvaaja on hyvin epäsäännöllinen useilla pelitilanteilla t, on evaluointifunktiota ef vastaava optimointiongelma hyvin hankala.

Tässä tutkielmassa käsiteltävät evaluointifunktiot voidaan esittää kaavoina, joihin sijoittamalla tiettyä pelitilannetta vastaavien muuttujien arvot, saadaan aritmeettisia lausekkeita. Täten tutkielmassa käsiteltyjen evaluointifunktioiden tef funktiot $test_{tef(t)}$ ovat jatkuvia kaikilla pelitilanteilla t lukuun ottamatta joitain

epäjatkuvuuskohtia (nollalla jakaminen). Sama pätee myös suurimmalle osalle evaluointifunktioista, jotka esitetään erilaisina laskukaavoina. Tässä tutkielmassa oletetaan, että optimoitavan evaluointifunktion ef funktion $test_{ef(t)}$ kuvaaja on pääosin jatkuva kaikilla pelitilanteilla t. Tämä tarkoittaa, että evaluointifunktion vakioiden muuttaminen vaikuttaa evaluointien arvoihin tasaisesti.

Tarkastellaan funktiota $power_{ef}$, joka mittaa kuinka hyvä evaluointifunktio ef on. Käytännössä usein evaluointifunktion hyvyys mitataan testaamalla miten evaluointifunktio pärjää pelissä. Funktio $power_{ef}$ voidaan toteuttaa esimerkiksi sellaisena, että se peluuttaa testattavaa evaluointifunktiota yhtä tai useampaa evaluointifunktiota vastaan ja pelien tuloksien mukaan antaa jonkin arvon. Tutkielmassa tehdyissä optimoinneissa on toteutettu funktio $power_{ef}$ juuri tällä tavalla. Toinen vaihtoehto olisi käyttää pelistä opittuja tietoja hyväksi ja testata miten hyvin evaluointifunktiot pelaavat.

Koska pelattavat pelit eivät ole aina samanlaisia, voi funktio $power_{ef}$ antaa eri arvoja samalle evaluointifunktiolle. Tämä on mahdollista esimerkiksi silloin, jos pelaaja arpoo siirtonsa useiden yhtä hyvien siirtojen väliltä tai evaluointifunktio antaa eri arvoja samoilla syötteillä. Funktion $power_{ef}$ satunnainen käyttäytyminen vaikeuttaa optimointiongelmaa. Tutkielmassa on suoritettu sekä optimointeja, joissa funktioon $power_{ef}$ on vaikuttanut sattuma, että optimointeja, joissa sattuman vaikutus on poistettu.

Funktio $power_{ef}$ ei voi antaa absoluuttista hyvyysarvoa evaluointifunktiolle, koska tämä arvo ei ole yksikäsitteisesti määritetty. Evaluointifunktiot pärjäävät pelissä eri tavalla erilaisissa tilanteissa ja erilaisia vastustajia vastaan. Siksi niitä ei voi laittaa täydelliseen paremmuusjärjestykseen. Tyydymme olettamaan, että funktio $power_{ef}$ antaa meille oikeat arvot ja pyrimme saamaan funktion käytännön toteutuksen antamaan mahdollisimman oikeita arvoja.

4.2.2 Optimointimenetelmiä

Merkitään $opt_{ef} : \mathbb{R}^n \rightarrow \mathbb{R}, opt_{ef}(x) = -power_{ef}(replace_{ef}(x))$ eli opt_{ef} on optimointiongelman funktio. Yleisesti ottaen pätee: mitä säännöllisempi optimoitava funktio on, sitä tehokkaampaa on optimointiongelman ratkaiseminen. Siksi kannattaa pyrkiä toteuttamaan funktio opt_{ef} mahdollisimman säännölliseksi. Funktion säännöllisyyteen vaikuttavat oleellisesti edellisessä luvussa esitellyn funktion $test_{ef(t)}$ säännöllisyys eri pelitilanteilla t, ja funktion $power_{ef}$ sään-

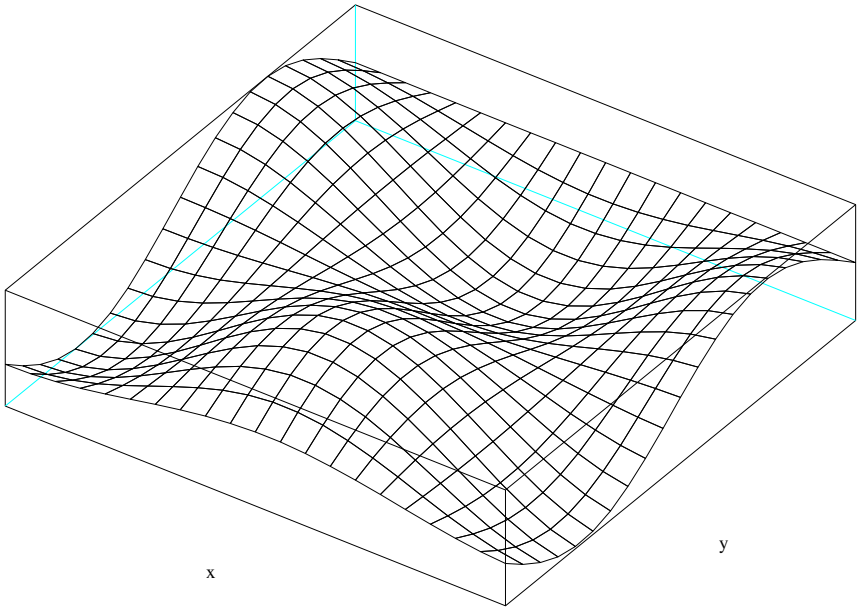
nöllisyys. Funktion $test_{ef(t)}$ säännöllisyys riippuu suoraan evaluointifunktiosta ef. Täten kannattaa valita mahdollisimman säännöllisiä evaluointifunktioita optimoitavaksi ja mahdollisesti muuttaa niitä säännöllisempään muotoon. Funktion $power_{ef}$ säännöllisyys taas riippuu sen toteutuksesta, joten toteutus kannattaa pyrkiä tekemään mahdollisimman säännölliseksi.

Funktio opt_{ef} gradientin laskeminen on usein mahdotonta. Vaikka funktion kuvaaja olisi säännöllinen, itse funktiota ei yleensä pystytä esittämään kaavana. Täten sen gradientin laskeminen derivointimenetelmin ei onnistu. Tästä syystä tutkielmassa rajoitutaan optimointimenetelmiin, jotka eivät vaadi gradientin laskemista.

Jos funktio $power_{ef}$ on toteutettu hyvin ja evaluointifunktio ef on tarpeeksi säännöllistä muotoa, voi funktion opt_{ef} kuvaaja muistuttaa jatkuvan funktion kuvaajaa. Funktion ei tarvitse olla jatkuva yleisessä matemaattisessa mielessä [Rud76, sivu 85] vaan riittää, että se on jatkuva johonkin tarkkuuteen rajatussa diskreetissä käytännön koordinaatistossa. Tässä koordinaatistossa voidaan sallia jopa joitain yksittäisiä epäjatkuvuuskohtia.

Intuitiivisesti jatkuvuus tarkoittaa funktion $opt_{ef} : \mathbb{R}^n \to \mathbb{R}$ kohdalla sitä, että sen kuvaaja muodostaa $n+1$ -ulotteisessa avaruudessa yhtenäisen alueen eikä irrallisia pisteitä (kuva 5). Jatkuva funktio on derivoituva, jos sillä on jokaisessa pisteessä derivaatta [Rud76, sivu 104]. Tämä tarkoittaa intuitiivisesti sitä, että funktion kuvaajan muodostama yhtenäinen alue on sileä eli siinä ei ole jyrkkiä kulmia (kuva 5). Derivoituville funktioille on paljon tehokkaampia optimointimenetelmiä kuin ainoastaan jatkuville funktioille. Tästä syystä funktiota opt_{ef} optimoitaessa on järkevä ensin tutkia minkälainen funktio se on. Jos funktio vaikuttaa derivoituvalta, voidaan käyttää esimerkiksi menetelmiä, joissa gradienttien arvoja yritetään muodostaa funktion arvojen perusteella [Gil81, sivut 127-133].

Jos funktio opt_{ef} on jatkuva, mutta ei ole derivoituva, on olemassa useita menetelmiä sen optimoinnille. Nämä menetelmät perustuvat yleensä funktioiden arvojen vertailuun ja johonkin heuristiikkaan [Gil81, sivut 93-98]. Ne ovat siten sovellettavissa myös tapauksiin, joissa optimoitava funktio ei ole edes jatkuva. Kuitenkin nämä optimointimenetelmät toimivat sitä paremmin, mitä säännöllisempi optimoitava funktio on. Jos funktio on niin epäsäännöllinen, että sen käyttäytyminen on lähes satunnaista, jää ainoaksi optimointikeinoksi pisteiden arpominen ja testaaminen.



Kuva 5: *Erään jatkuvan ja derivoituvan funktion* $f : \mathbb{R}^2 \rightarrow \mathbb{R}$ *kuvaaja rajatulla alueella.*

4.2.3 Tutkielman optimointiympäristö

Tutkielmassa pyritään optimoimaan funktiota opt_{ef}. Tämä tarkoittaa, että evaluointifunktiolle ef etsitään parempia vakioiden arvoja. Tutkielman kannalta riittää löytää alkuarvoja parempia vakioiden arvoja. Parempien arvojen löytäminen merkitsee tietokoneen pelitason paranemista. Tämä voidaan tulkita siten, että tietokone oppii paremmaksi pelaajaksi.

Tutkielmassa on toteutettu yksinkertainen optimointialgoritmi, joka perustuu evaluointifunktioiden vertailuun. Gradienttimenetelmän (*gradient method* tai *steepest descent*) [Lue84, sivut 214-215] ideaa on sovellettu toteutettuun algoritmiin. Mitään optimointiohjelmistoja ei ole käytetty, koska ne ovat yleisesti ottaen suunniteltu huomattavasti opt_{ef}:ää säännöllisempien funktioiden optimointiin.

Gradienttimenetelmä on optimointimenetelmä, jossa funktion gradientin avulla voidaan löytää funktion lokaali minimi. Kuvassa 5 on erään kaksiulotteisen funktion rajoitettu kuvaaja. Tämän funktion gradientti ilmoittaa kussakin pisteessä suurimman kasvun suunnan [Gil81, sivu 50]. Gradienttimenetelmä etenee siten, että ensin lasketaan gradientti jossain pisteessä. Tämän jälkeen liikutaan gradientin vastaiseen suuntaan tietyn askeleen verran. Uudessa pisteessä lasketaan jälleen gradientti ja siirrytään sen vastaiseen suuntaan askeleen verran. Askeleen

pituus voi muuttua algoritmin edetessä. Jos optimoitava funktio täyttää tietyt säännöllisyysehdot ja askeleen pituus on valittu sopivasti, gradienttimenetelmällä hakeudutaan kohti lokaalia minimiä [Lue84, sivu 215]. Konvergenssin nopeuteen vaikuttaa suuresti askeleen pituus. Kuvassa 5 gradienttimenetelmä etenisi jostain pisteestä askeleittain kohti lokaalia minimiä. Jos askeleet otetaan joka kerta gradientin suunnan sijaan jonkin akselin suuntaan, kutsutaan menetelmää stokastiseksi.

Tutkielmassa toteutettu optimointialgoritmi etenee siten, että ensin sille annetaan optimoitava evaluointifunktio. Tämän jälkeen algoritmi muuttaa evaluointifunktion vakioiden arvoja yksi kerrallaan ja testaa miten muutokset vaikuttavat evaluointifunktion suoritukseen. Jos suoritus paranee, vaihdetaan vakio testattuun vakioon. Jos suoritus ei parane, pidetään evaluointifunktio ennallaan. Askelmuuttuja määrää minkä verran vakioita muutetaan. Se voi pysyä samana (yleensä pienenä) koko algoritmin suorituksen ajan tai se voi pienentyä algoritmin edetessä, jolloin algoritmi toimii hieman simuloidun jäähdytyksen (*simulated annealing*) periaatteiden mukaisesti [Laa87, sivut 7-12].

Askelmuuttujan väheneminen voi olla joko lineaarista tai prosentuaalista. Algoritmi pysähtyy kun askel on riittävän lyhyt eli askelmuuttuja on määrätyn rajan alapuolella tai kun tietty määrä päivityskierroksia on täynnä. Algoritmi voi toimia niin, että se ryhtyy suoraan muuttamaan evaluointifunktion vakioita. Toinen tapa on nollata kaikki vakiot ja määrätä ensimmäisen vakion arvo. Tällöin ikään kuin asetetaan lähtötaso, jonka mukaan vakiot määräytyvät (edellyttäen, että niiden keskinäinen suhde pysyy samana optimipisteessä).

Optimointialgoritmin jokaisella suorituskierroksella luodaan kaksi uutta evaluointifunktiota muuttamalla edellisen kierroksen evaluointifunktion tietyn vakion arvoa askelmuuttujan verran suuremmaksi tai pienemmäksi. Tämän jälkeen vertaillaan kahta juuri luotua evaluointifunktiota ja edellisen kierroksen evaluointifunktiota ja valitaan näistä paras. Tällä tavalla valittuna oleva evaluointifunktio on aina vähintään yhtä hyvä kuin aiemmat evaluointifunktiot ja optimoinnin edetessä päädytään aina yhtä hyviin tai parempiin evaluointifunktioihin. Evaluointifunktioiden keskinäinen paremmuus mitataan joko peluuttamalla niitä keskenään tai vertaamalla niitä määrättyyn evaluointifunktioon.

Optimoinnin toteutus vastaa stokastista gradienttimenetelmää. Vakioiden määrä kertoo avaruuden ulottuvuuden, johon optimoitavan funktion arvot muodostavat hypertason. Gradientin laskemisen mahdottomuuden takia optimoitavan

funktion gradienttia arvioidaan laskemalla funktion arvo tietyn askeleen päässä akselin suunnassa ja akselin vastaisessa suunnassa. Näin saadaan aikaan karkea arvio siitä, kasvaako vai väheneekö funktion arvo akselin suuntaan liikuttaessa. Lisänä gradienttimenetelmään on se, että aina ei välttämättä muuteta vakioita eli liikuta hypertasolla. Jos jonkin akselin suhteen nykyinen piste on parempi kuin molempiin suuntiin tehdyt askeleet, ei akselin suuntaan liikuta.

Tutkielman optimointiympäristö on toteutettu Java-luokkana (liite 1), joka saa optimoitavan evaluointifunktion parametrinaan ja optimoi sen ennalta määriteltyjen parametrien mukaisesti. Optimoinnista kootaan raportti, josta voidaan myöhemmin tarkastella optimoinnin etenemistä. Tarkempia tietoja optimoinnin toteutuksesta löytyy tutkielman sovelluksen WWW-sivulta [ImpOthello] ja esimerkkejä optimoinnista koostetuista raporteista voi katsoa tutkielman tulosten WWW-sivulta [Sii02]

4.3 Optimoinnin käytännön kokemuksia

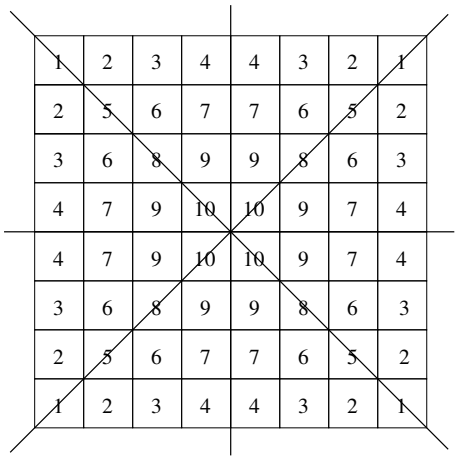
Edellä esitellyn optimointiympäristön avulla on pyritty parantamaan tiettyä evaluointifunktiota. Tässä luvussa esitellään aluksi optimoitava evaluointifunktio ja se miten optimointitestit on toteutettu. Tämän jälkeen esitellään testien tulokset.

4.3.1 Optimoitava Othellon evaluointifunktio

Optimoitavan evaluointifunktion pohjana on toiminut evaluointifunktio *sum*, joka laskee pelilaudalla olevat pelaajien kiekot ja palauttaa niiden erotuksen. Evaluointi on negatiivinen, jos valkoisella on enemmän kiekkoja, ja positiivinen, jos mustalla on enemmän kiekkoja. Koska pelilaudan tilanne on koodattu niin, että evaluointifunktion muuttujan $[i, j]$ arvo on 0, 1 tai -1, sen mukaan minkä värinen kiekko ruudussa (i, j) on, voidaan evaluointifunktio *sum* esittää muodossa $\sum_{0 \leq i,j \leq 7} [i, j]$.

Optimoitava evaluointifunktio saadaan lisäämällä evaluointifunktioon *sum* painokertoimia. On luontevaa olettaa, että eri ruuduilla on erilainen painoarvo. Esimerkiksi kulmaruudut ovat Othellossa erityisen arvokkaita, koska niitä ei voi vastustaja enää kaapata itselleen [Mor93, sivu 2]. Evaluointifunktio, joka laskee ruutujen arvot yhteen painottaen kulmaruutuja, toimii paremmin kuin evaluointifunktio, joka vain laskee ruutujen arvot yhteen. Painotettu summa voidaan esit-

tää muodossa $\sum_{0\leq i,j\leq 7} a_{i,j} \times [i,j]$, jossa $a_{i,j}$, $0 \leq i,j \leq 7$, ovat painokertoimia. Koska Othellon pelilauta on symmetrinen neljän akselin suhteen, samoja painokertoimia voidaan käyttää useille ruuduille. Tämä on suositeltavaa, koska se vähentää evaluointifunktiossa olevien vakioiden määrää. Tällöin optimointitehtävä helpottuu ja evaluointifunktion arvojen laskeminen nopeutuu.



Kuva 6: *Othellon pelilaudan symmetria-akselit ja ruutujen* 10 *eri painokerrointa.*

Tutkielmassa optimoitavaa evaluointifunktiota kutsutaan nimellä *weight*10. Kuvassa 6 on tyhjän pelilaudan symmetria-akselit ja ruutujen eri painokertoimet. Siitä nähdään, että jos tarkastellaan vain pelilautaa ja käytetään hyväksi symmetriaa, riittää 10 painoarvoa kattamaan koko laudan. Taulukossa 1 on yleisen kuvaan 6 perustuvan evaluointifunktion kaava. Evaluointifunktio *weigth*10 saadaan määräämällä $a_i = 1$, kaikilla $i = 1, \ldots, 10$.

$ef =$	$a_1 \times ([0,0] + [0,7] + [7,0] + [7,7]) + a_2 \times ([1,1] + [1,6] + [6,1] + [6,6]) +$
	$a_3 \times ([2,2] + [2,5] + [5,2] + [5,5]) + a_4 \times ([3,3] + [3,4] + [4,3] + [4,4]) +$
	$a_5 \times ([1,0] + [0,1] + [6,0] + [0,6] + [7,1] + [1,7] + [7,6] + [6,7]) +$
	$a_6 \times ([2,0] + [0,2] + [5,0] + [0,5] + [7,2] + [2,7] + [7,5] + [5,7]) +$
	$a_7 \times ([3,0] + [0,3] + [4,0] + [0,4] + [7,3] + [3,7] + [7,4] + [4,7]) +$
	$a_8 \times ([2,1] + [1,2] + [5,1] + [1,5] + [6,2] + [2,6] + [6,5] + [5,6]) +$
	$a_9 \times ([3,1] + [1,3] + [4,1] + [1,4] + [6,3] + [3,6] + [6,4] + [4,6]) +$
	$a_{10} \times ([3,2] + [2,3] + [4,2] + [2,4] + [5,3] + [3,5] + [5,4] + [4,5])$

Taulukko 1: *Kuvaan 6 perustuvan evaluointifunktion kaava.*

Evaluointifunktio *weight*10 on oleellisesti sama kuin funktio $\sum_{0\leq i,j\leq 7} a_{i,j} \times [i,j]$, missä painokertoimet $a_{i,j}$, $0 \leq i,j \leq 7$, ovat ykkösiä. Tästä huolimatta evaluointifunktiot muodostavat eri joukot E_{ef}. Ne sisältävät eri määrän vakioita (10 ja 64), jotka

tässä tapauksessa ovat painokertoimia. Joukkoon $E_{weight10}$ kuuluvat kaikki evaluointifunktiot, joissa taulukossa 1 esitellyn evaluointifunktion painokertoimina ovat mitkä tahansa reaaliluvut. Optimointitehtävä on nyt mahdollisimman hyvien painokerrointen etsiminen.

Tässä tutkielmassa on rajoituttu ainoastaan funktion *weight*10 optimointiin, jotta tutkielman aihepiiri ei paisuisi liian suureksi. Lisäksi kiinnostuksen kohteena ovat olleet kuinka vahvaksi vain ruutujen painokertoimiin perustuvan evaluointifunktion voi kehittää ja konvergoivatko painokerrointen arvot tai keskinäinen suhde joitain tiettyjä arvoja kohti.

4.3.2 Optimointitestien toteutus

Tutkielmassa toteutetussa optimointiympäristössä vastaa optimointiongelman funktiota $power_{ef}$ periaatteessa evaluointifunktioiden vertailu. Se ilmoittaa kuinka hyvä evaluointifunktio on suhteessa toiseen. Vertailussa pelattavat pelit voidaan toteuttaa kahdella eri tavalla:

1. Pelaajat arpovat yhtä hyvien siirtojen kohdalla tekemänsä siirron.
2. Pelaajat siirtävät aina ensimmäisen löydetyn parhaan siirron.

Molempia vertailuja on käytetty tutkielmassa tehdyissä optimointitesteissä. Suoritetuista optimoinneista käytetään nimityksiä satunnaiset optimoinnit (käyttävät vertailua 1) ja ei-satunnaiset optimoinnit (käyttävät vertailua 2).

Ensimmäinen toteutustapa on parempi siinä mielessä, että ohjelma ei pelaa aina samalla tavalla. Tällöin esimerkiksi vastustaja ei voi kerran voittaessaan pelata aina samalla tavalla ja voittaa. Tällainen pelitapa vaikuttaa ihmisvastustajan mielestä järkevämmältä ja inhimillisemmältä. Toisaalta, kun pelaaja arpoo parhaiden siirtojen välillä, ei pelin lopputulos ole ennalta määrätty. Tällöin myöskään vertailun tulos ei ole ennalta määrätty vaan siihen vaikuttaa sattuma. Vertailun tuloksen satunnaisuus hankaloittaa optimointitehtävää. Satunnaisuutta voidaan vähentää peluuttamalla evaluointifunktioita vastakkain useita kertoja. Tällöin samalla kuitenkin vertailuun kuluva aika kasvaa.

Toisessa toteutustavassa ohjelma tekee jokaisella pelikerralla samassa tilanteessa saman siirron. Tällöin pelit tulevat ennalta määrätyiksi ja sattuman vaikutus evaluointifunktioiden vertailun tulokseen häviää. Koska pelit etenevät aina samalla

tavalla, riittää ennalta määrättyjä vertailuja suoritettaessa pelata kaksi peliä: vertailtavat evaluointifunktiot pelaavat sekä mustilla, että valkoisilla kiekoilla. Tästä syystä vertailuihin kuluva aika on ennalta määrättyjen pelien tapauksessa melko lyhyt.

Ennalta määrätyt pelit eivät anna kovin monipuolista tietoa evaluointifunktioiden suoriutumisesta. Ennalta määrätty peli voi olla esimerkiksi sellainen, että huonompi evaluointifunktio sattuu voittamaan sen. Tällöin optimoinnissa suositaan huonompaa evaluointifunktiota. Satunnainen toteutustapa antaa yleensä ennalta määrättyä tapaa oikeamman vertailutuloksen, mutta vie enemmän aikaa.

Tutkielmassa on käytetty kahta intuitiivista mittaria, jotka antavat numeroarvon vertailtavien evaluointifunktioiden suoriutumiselle:

1. Voittoprosentti eli voitettujen pelien määrän suhde pelattuihin peleihin.
2. Pelikiekkoprosentti eli kaikkien pelien lopputilanteiden osalta pelaajan pelikiekkojen määrän suhde kaikkiin pelikiekkoihin.

Voittoprosentti laskee kuinka monta prosenttia peleistä vertailtava evaluointifunktio voittaa. Tasapelit lasketaan puolikkaiksi voitoiksi molemmille. Mittari antaa tilastollisen tavan vertailla evaluointifunktioiden suoriutumista, mutta ongelmana on että hyvän vertailun aikaan saamiseksi on pelattava paljon pelejä. Tulos on sitä tarkempi, mitä enemmän pelattuja pelejä on. Pelien määrän kasvattaminen kasvattaa kuitenkin myös evaluointifunktioiden vertailuun kuluvaa aikaa.

Pelikiekkoprosentti ilmoittaa kuinka monta prosenttia pelikiekoista vertailtava evaluointifunktio sai. Tämän mittarin hyvänä puolena on, että se on tarkempi kuin voittoprosentti. Yhden pelin jälkeen pelikiekkoprosenttiin perustuva mittari voi antaa 65 erilaista arvoa, kun taas voittoprosenttiin perustuva mittari voi antaa vain 3 erilaista arvoa. Täten saman tarkkuuden saavuttamiseksi riittää pelikiekkoprosenttia käytettäessä vähempi määrä pelejä.

Pelikiekkoprosentti ei heittelehdi niin paljon kuin voittoprosentti. Jos pelaaja esimerkiksi häviää kaikki pelinsä, on voittoprosentti 0%, mutta pelikiekkoprosentti voi olla lähellä 50%:a riippuen siitä kuinka niukasti pelaaja on pelit hävinnyt. Täten sattuman vaikutus pelikiekkoprosenttiin on pienempi kuin voittoprosenttiin. Toisaalta jos ajatellaan pelin lopputuloksen olevan oleellisinta, voi pelikiekkoprosentti antaa turhan tasaväkistä tai jopa väärää tietoa evaluointifunktioiden keskinäisestä paremmuudesta. Yleensä pelikiekkoprosentti on lähempänä 50%:a kuin

voittoprosentti, koska pelaajien voitot eivät yleensä ole sellaisia, että toisen pelaajan kaikki pelikiekot häviävät pelilaudalta. Tällöin pelikiekkoprosentti tulkitsee pelaajat tasaväkisemmäksi kuin ne ehkä käytännössä ovatkaan. Periaatteessa pelikiekkoprosentti voi ilmoittaa eri pelaajan paremmaksi kuin voittoprosentti. Jos toinen pelaajista voittaa yli puolet peleistä niukasti ja häviää loput selkeästi, määräävät voitto- ja pelikiekkoprosentit eri pelaajat paremmiksi.

Tässä tutkielmassa suoritetuissa käytännön optimointitesteissä on käytetty kahta erilaista vertailua. Toisessa evaluointifunktiot pelaavat 100 peliä vastakkain siten, että ne arpovat parhaiden siirtojen väliltä jonkin. Vertailun tulos mitataan voittoprosentteina. Näin saadaan kohtalainen arvio pelaajien keskinäisestä paremmuudesta suhteellisen pienellä ajankäytöllä. Vertailujen tuloksissa on kuitenkin satunnaisuutta, joka vaikeuttaa optimointitehtävää.

Toisessa vertailussa pelataan 2 peliä siten, että pelaajat siirtävät aina ensimmäisenä löydetyn parhaan siirron. Vertailujen tulokset lasketaan pelikiekkoprosentteina. Pelit ovat ennalta määrättyjä, joten vertailun tulokseen ei vaikuta sattuma. Lisäksi vertailuun kuluva aika on vain noin 2% ensiksi mainittuun vertailuun kuluvasta ajasta. Haittapuolena vertailussa on, että sen tulokset voivat olla virheellisiä.

4.3.3 Satunnaisen funktion optimoinnin testejä

Ensimmäinen suoritettu optimointitesti oli evaluointifunktion *weight*10 painoarvojen parantaminen satunnaiseen vertailuun perustuvin optimointimenetelmin. Testissä aloitettiin evaluointifunktion *weight*10 toisesta painokertoimesta (taulukossa 1 a_2) ja muutettiin sitä aina askelmuuttujan verran. Askeleen pituus oli aluksi 2 ja sitä pienennettiin joka kierroksella $0,1$:llä. Painoarvoa muutettiin 20 kierroksen verran, jonka jälkeen askeleen pituus oli nolla. Tämän jälkeen siirryttiin seuraavaan painoarvoon. Näin meneteltiin jokaisen jäljellä olevan painoarvon kohdalla (taulukossa 1 painokertoimet $a_2, \ldots, a_{10}$). (Katso ensimmäisen testin yhteenveto taulukosta 2.)

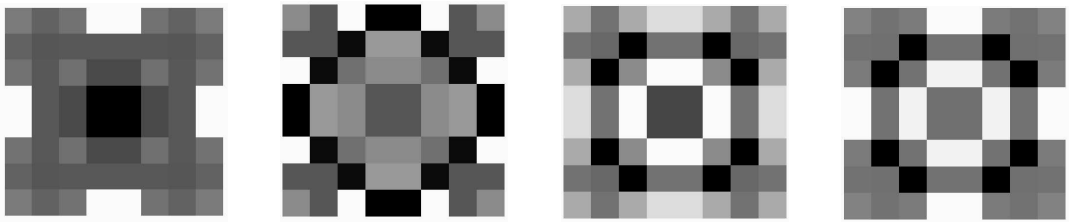
Menetelmässä painoarvojen muutos voi olla korkeintaan 21, koska

$$\sum_{i=0}^{19}(2 - (0,1 \cdot i)) = \sum_{i=1}^{20} 0,1 \cdot i = 0,1 \cdot \frac{20(1+20)}{2} = 21.$$

Täten jokainen muutettu painoarvo on välillä $[-20, 22]$. Samoin painoarvot muuttuvat aina vähintään $0,1$ askelin, joten painoarvot ovat $0,1$:n monikertoja.

Ennen testejä mitattiin, että yhteen optimointimenetelmän askeleeseen meni aikaa noin 40 sekuntia. Tästä syystä askelten lukumäärä painokerrointa kohti päätettiin pitää kohtalaisen pienenä. Koska jokaista muutettavaa 9:ää painokerrointa kohti suoritettiin 20 askelta, kului yhden testin suorittamiseen aikaa noin $9 \cdot 20 \cdot 40s = 7200s = 2h$. Optimointitesti toistettiin neljä kertaa samanlaisena, jotta satunnaisuuden vaikutusta voitiin jollain tavalla arvioida. Optimointien todelliset ajat vaihtelivat noin kahdesta tunnista kahteen ja puoleen tuntiin.

Tehtyjen testien tuloksina saadut evaluointifunktiot numeroitiin juoksevasti alkaen nollasta. Optimoinnin tuloksena saadun evaluointifunktion nimi muodostettiin juoksevasta numerosta ja optimoitavan evaluointifunktion nimestä kaavalla *evaluointifunktio.numero*. Ensimmäisen optimointitestin tulokset olivat evaluointifunktiot *weight*10.0, *weight*10.1, *weight*10.2 ja *weight*10.3. Ne on visualisoitu kuvassa 7 kuvaamalla pelilaudan ruutujen painoarvot. Tämä on toteutettu siten, että pienin painoarvo on musta, suurin valkoinen ja muut painoarvot harmaan sävyjä tältä väliltä. Täten painoarvojen keskinäiset suhteet skaalautuvat samoiksi.



*weight*10.0 *weigth*10.1 *weight*10.2 *weigth*10.3

Kuva 7: *Ensimmäisen optimointitestin tuloksena saadut evaluointifunktiot.*

Ensimmäisen testin ideana oli pitää evaluointifunktion *weight*10 ensimmäinen painoarvo ennallaan ja muuttaa loppuja painoarvoja niin, että ne sijoittuisivat (lähes) optimaalisesti ensimmäisen suhteen. Kuvasta 7 nähdään, että painoarvot eivät konvergoineet tiettyjä arvoja kohti, vaan ne päätyivät eri lukemiin eri optimointikerroilla. Parannusta alkuperäiseen kuitenkin tapahtui evaluointifunktioiden suoriutumisen osalta jokaisella optimointikerralla, kuten myöhemmin esiteltävästä taulukosta 4 nähdään.

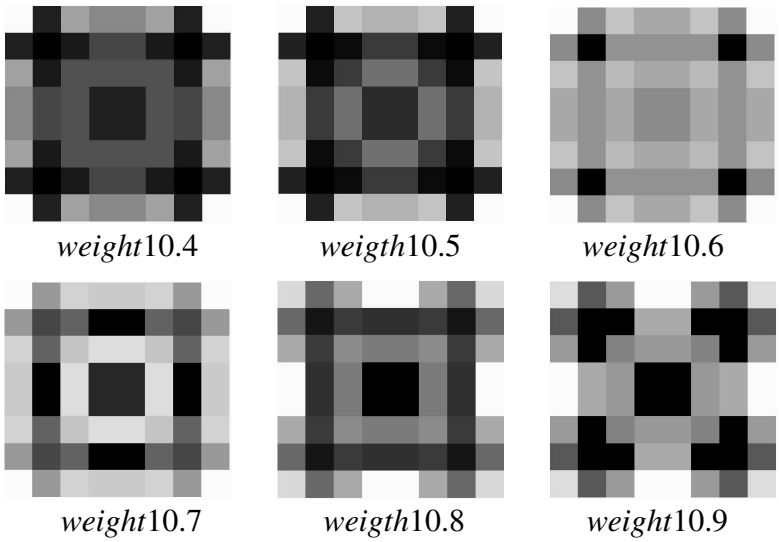
Toisessa testissä pyrittiin vähentämään satunnaisuuden vaikutusta. Tämä tehtiin kasvattamalla ensimmäinen painoarvo 1:stä 20:een, askeleen alkuarvo 2:sta 10:een ja askeleen pienennys $0,1$:stä 1:een. Lisäksi kaikki painoarvot, paitsi ensimmäinen, asetettiin nollaksi. Muut optimoinnin parametrit olivat samoja kuin ensimmäisessä testissä. Lähtökohtana oli oletus, että suuriin painokerrointen muu-

lähtökohta	Optimoidaan evaluointifunktiota *weight*10.
alustus	Ensimmäinen paino arvo muuttumaton, loput muutetaan.
askeleen alkuarvo	2
askeleen muutos	Joka kierroksella 0, 1 pois.
optimointiaika	Yhteen optimointiin kului 2-2,5 h.
toistot	Testi toistettiin 4 kertaa.
tulos	Optimointi paransi funktioita, konvergointia ei tapahtunut.

Taulukko 2: *Ensimmäisen testin yhteenveto.*

toksiin sattuma vaikuttaa vähemmän kuin pieniin muutoksiin. Kun evaluointifunktiot antavat selkeästi eri arvoja, on niiden välisen paremmuuden määrittäminen helpompaa.

Toisessa testissä haettiin painokertoimille yksi kerrallaan optimaalista arvoa lähellä olevaa arvoa. Koska käsittelemättömät painokertoimet olivat nollia, eivät ne vaikuttaneet evaluointifunktioiden arvoihin. Täten kukin painokerroin hakeutui lähelle optimipistettä suhteessa edellisiin ja lopputuloksena oli lähellä optimia olevat painokertoimet. Koska askeleet olivat kokonaislukuja, painokertoimet muuttuivat reilusti kerrallaan ja evaluointifunktioiden väliseen vertailuun ei vaikuttanut sattuma niin suuresti. Toisessa testissä toistettiin optimointi 6 kertaa. Yhteen testiin meni aikaa noin 57 minuuttia. (Katso toisen testin yhteenveto taulukosta 3)



Kuva 8: *Toisen optimointitestin tuloksena saadut evaluointifunktiot.*

Kuvassa 8 on visualisoitu toisen satunnaisen funktion optimoinnin tuloksena saadut evaluointifunktiot. Kuvasta nähdään, että edelleenkin painoarvot ovat päätyneet hyvin erilaisiin lukuihin. Täten satunnaisuuden vaikutus on yhä suuri. Jos vertailu olisi aina kahta evaluointifunktiota vertailtaessa ilmoittanut saman parhaaksi, olisivat kaikki optimoinnit päätyneet samaan evaluointifunktioon.

lähtökohta	Poistetaan ensimmäisen testin satunnaisuutta.
alustus	Ensimmäinen paino arvo 20, loput nollataan.
askeleen alkuarvo	10
askeleen muutos	Joka kierroksella 1 pois.
optimointiaika	Yhteen optimointiin kului noin 57 min.
toistot	Testi toistettiin 6 kertaa.
tulos	Optimointi paransi funktioita, konvergointia ei tapahtunut.

Taulukko 3: *Toisen testin yhteenveto.*

Tulosten satunnaisuudesta huolimatta optimoinnin tuloksena saatiin paljon parempia evaluointifunktioita kuin alkuperäinen. Tämä voidaan todeta taulukosta 4, johon on kerätty ensimmäisen ja toisen optimointitestin tulosten vertailuja. Vertailuissa on pelattu 100 satunnaista peliä. Pelaajat (evaluointifunktiot *weight*10 ja *weight*10.[0 – 9]) on merkitty taulukon ensimmäiselle riville ja sarakkeelle, mutta sana *weight* on jätetty nimien alusta pois tilan säästämiseksi. Taulukon luvut kertovat ensimmäisen pelaajan (rivit) voittoprosentin sen pelatessa toista pelaajaa (sarakkeet) vastaan.

Taulukosta 4 voidaan nähdä vertailujen satunnaisuus, sillä jokaista evaluointifunktiota on verrattu kaksi kertaa jokaiseen toiseen evaluointifunktioon. Vertailun tulos ei ole riippuvainen vertailtavien järjestyksestä, joten esimerkiksi evaluointifunktioiden *weight*10.1 ja *weight*10.7 vertailun sekä evaluointifunktioiden *weight*10.7 ja *weight*10.1 vertailun tulosten summan tulisi olla 100%. Näin ei kuitenkaan ole, vaan summa on $78,5\% + 25,0\% = 103,5\%$. Ylimääräinen $3,5\%$ selittyy sillä, että vertailut antoivat niissä vaikuttavan satunnaisuuden takia hieman todellista paremman voittoprosentin. Tulosten satunnaisuuden takia taulukon 4 tulokset ovat vain suuntaa antavia.

Taulukon tuloksia tarkasteltaessa kaksi evaluointifunktiota näyttävät olevan muita parempia: *weight*10.4 ja *weight*10.5. Nämä molemmat voittavat vertailussa kaikki muut evaluointifunktiot paitsi yhden. Evaluointifunktio *weight*10.4 häviää ver-

	10	**10.0**	**10.1**	**10.2**	**10.3**	**10.4**	**10.5**	**10.6**	**10.7**	**10.8**	**10.9**
10	-	36,5	28,5	27,0	15,5	15,5	15,0	22,5	30,5	25,0	21,5
10.0	57,5	-	26,0	12,0	5,5	21,0	6,0	42,0	6,0	39,0	15,5
10.1	75,5	77,5	-	54,5	39,5	17,0	21,0	60,0	78,5	68,5	47,0
10.2	79,0	86,5	53,0	-	70,5	41,0	21,5	25,5	71,0	57,5	38,5
10.3	87,0	94,0	59,0	28,5	-	15,0	18,0	30,0	68,5	53,0	37,0
10.4	84,0	85,5	75,5	66,0	71,5	-	46,0	64,0	87,5	72,5	84,5
10.5	84,5	87,5	74,5	80,5	75,0	57,5	-	41,0	69,5	65,0	59,0
10.6	77,5	71,5	37,0	76,5	77,0	27,5	51,0	-	90,0	69,0	64,5
10.7	66,0	99,0	25,0	26,0	42,5	14,5	25,0	9,5	-	49,5	15,0
10.8	71,0	64,0	46,5	50,5	49,0	31,5	37,0	35,5	54,0	-	21,0
10.9	80,5	87,5	58,0	50,5	78,5	35,0	37,0	36,0	89,0	73,5	-

Taulukko 4: *Evaluointifunktioiden välisten vertailujen voittoprosentteja.*

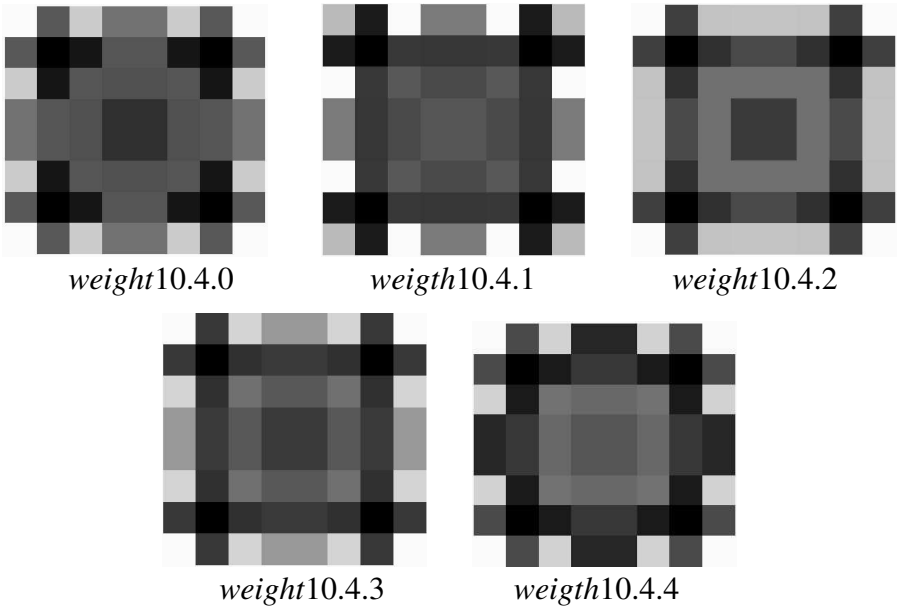
tailussa vain evaluointifunktiolle *weight*10.5 ja *weight*10.5 vain evaluointifunktiolle *weight*10.6. Toisaalta evaluointifunktio *weight*10.6 häviää vertailussa evaluointifunktiolle *weight*10.4, joten vertailut eivät täytä transitiivisuusehtoa (jos $a < b$ ja $b < c$ niin $a < c$) [Rud76, sivu 25]. Tämä ominaisuuden takia optimoinnissa voidaan päätyä silmukkaan, jossa koko ajan saadaan vertailujen mukaan edellistä evaluointifunktiota parempi evaluointifunktio, mutta käytännössä vaihdellaan vain joitain evaluointifunktioita toisikseen.

Taulukosta nähdään myös, että keskinäisen vertailun tulos voi olla harhaan johtava. Kun vertaillaan evaluointifunktioita *weight*10.7 ja *weight*10.0 voidaan saada niinkin korkea voittoprosentti kuin $99,0\%$ kuten taulukossa 4. Tämä ei kuitenkaan tarkoita, että evaluointifunktio olisi ylivoimaisen hyvä, sillä *weight*10.7 häviää vertailussa kaikille muille paitsi evaluointifunktioille *weight*10 ja *weight*10.0. Täten sitä ei voi pitää korkeasta voittoprosentistaan huolimatta kovin hyvänä evaluointifunktiona. Se vain sattuu olemaan sellainen, että se pärjään erittäin hyvin evaluointifunktiota *weight*10.0 vastaan.

Taulukosta 4 nähdään selvästi, että jokainen optimoinnin tuloksena saatu evaluointifunktio on parempi kuin alkuperäinen optimoitava evaluointifunktio. Tämä tulos motivoi kolmatta ja viimeistä testiä, joka tehtiin satunnaisen funktion optimoinnilla. Kolmannen testin optimoinnin aloitusfunktioksi valittiin paras aiempien optimointien tuloksista. Taulukon 4 tuloksien mukaan paras evaluoin-

tifunktio on *weight*10.4. Sillä on yhtä suuri voittoprosenttien summa kuin evaluointifunktiolla *weight*10.5 (694, 0%), mutta 3% pienempi häviöprosenttien summa (275,5%).

Kolmannessa testissä optimoinnin lähtökohdaksi otettiin funktio *weight*10.4 ja sitä ryhdyttiin parantamaan painokerroin kerrallaan aloittaen askelpituudesta 2 ja pienentäen sitä kierroksittain luvulla 0, 1 aina nollaan asti. Testi toistettiin 5 kertaa. Kukin optimointi kesti noin kaksi tuntia. (Katso kolmannen testin yhteenveto taulukosta 5.) Testin tuloksina saadut evaluointifunktiot on visualisoitu kuvassa 9.



Kuva 9: *Kolmannen optimointitestin tuloksena saadut evaluointifunktiot.*

Kuvasta 9 nähdään, että painoarvot eivät edelleenkään konvergoineet kohti samoja arvoja. Eri evaluointifunktioiden visualisoinneissa on tiettyä samankaltaisuutta, mutta tämä johtuu siitä, että optimointi oli periaatteessa evaluointifunktion *weight*10.4 hienosäätöä. Painoarvojen askelmuutos oli sen verran pieni ja askelia sen verran vähän, että niiden kokonaismuutos jäi melko pieneksi. Alkuperäisten painoarvojen keskinäiset suhteet eivät voineet muuttua täysin.

Taulukkoon 6 on koottu evaluointifunktion *weight*10.4 ja sen optimoinnin tulosten vertailuja. Taulukko on samanmuotoinen kuin taulukko 4 paitsi että se sisältää sekä vertailujen voittoprosentin (ensimmäinen luku) että pelikiekkoprosentin (toinen luku). Taulukon evaluointifunktiot ovat sen verran tasaväkisiä, että tarkemman vertailun aikaan saamiseksi niitä on verrattu kahdella eri mittarilla. Tau-

lähtökohta	Hienosäädetään funktiota *weight*10.4.
alustus	Kaikkia painoarvoja hienosäädetään.
askeleen alkuarvo	2
askeleen muutos	Joka kierroksella 0, 1 pois.
optimointiaika	Yhteen optimointiin kului noin 2 tuntia.
toistot	Testi toistettiin 5 kertaa.
tulos	Optimointi ei parantanut funktioita, lievää konvergointia.

Taulukko 5: *Kolmannen testin yhteenveto.*

lukon ensimmäiseltä numeroriviltä ja -sarakkeelta nähdään, että evaluointifunktiot *weight*10.4.0, *weight*10.4.1 ja *weight*10.4.2 häviävät vertailussa alkuperäiselle evaluointifunktiolle *weight*10.4. Optimointi ei siis niiden osalta tuottanutkaan alkuperäistä parempaa evaluointifunktiota. Optimointi etenee kuitenkin niin, että evaluointifunktiota muutetaan vain sellaiseen suuntaan, että se on kahdenkeskisessä vertailussa parempi kuin edeltäjänsä. Evaluointifunktioiden huonontuminen selittyy kuitenkin vertailuissa vaikuttavalla satunnaisuudella ja sillä, että vertailut eivät ole transitiivisia.

	10.4	**10.4.0**	**10.4.1**	**10.4.2**	**10.4.3**	**10.4.4**
10.4	-	65, 0/60, 0	61, 5/56, 1	58, 5/54, 7	46, 0/47, 2	37, 0/45, 2
10.4.0	40, 0/41, 4	-	53, 5/50, 1	37, 0/49, 3	69, 0/60, 5	38, 5/46, 1
10.4.1	36, 0/44, 2	48, 0/49, 3	-	60, 5/52, 7	59, 5/52, 9	57, 0/52, 8
10.4.2	46, 0/48, 1	56, 0/51, 3	46, 5/46, 3	-	38, 5/46, 0	16, 0/40, 5
10.4.3	66, 5/56, 1	31, 0/38, 1	35, 0/48, 8	62, 5/53, 5	-	63, 0/51, 8
10.4.4	57, 0/51, 7	59, 0/53, 4	45, 0/50, 0	78, 5/58, 1	41, 5/48, 5	-

Taulukko 6: *Evaluointifunktion weight*10.4 *ja sen optimoinnin tuloksien vertailu.*

Evaluointifunktiot *weight*10.4.3 ja *weight*10.4.4 voittavat vertailussa alkuperäisen evaluointifunktion *weight*10.4. Niiden osalta voidaan siis sanoa, että optimointi on tuottanut paremman tuloksen. Ne kuitenkin häviävät toisille taulukossa oleville evaluointifunktioille, joten nekään eivät yksikäsitteisesti ole parantuneet. Taulukosta on vaikea valita parasta tai huonointa evaluointifunktiota, koska vertailujen tulokset menevät ristiin. Alkuperäinen evaluointifunktio ei erotu muita huonompana, joten optimointi ei oleellisesti parantanut evaluointifunktioita.

Tulos voi johtua huonosta parametrien valinnasta tai se voi olla merkki käytettyjen optimointimenetelmien ylärajan lähestymisestä. Satunnaisuus yksinään takaa sen, että tiettyä tarkkuutta parempaan tulokseen ei päästä. Raja tulee vastaan, kun satunnaisuuden vaikutus on niin suuri, että menetelmä toimii täysin satunnaisesti. Seuraavassa luvussa käsitellään tarkemmin tapausta, jossa satunnaisuus on poistettu.

4.3.4 Ei-satunnaisen funktion optimoinnin testejä

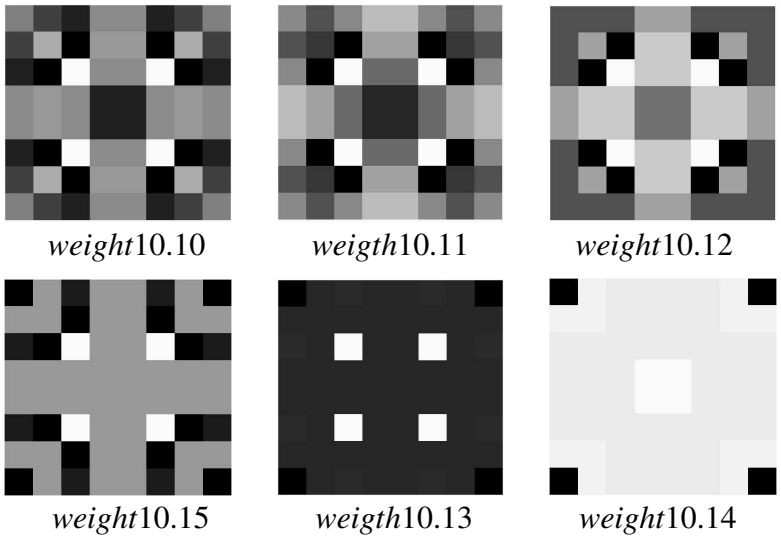
Tässä luvussa käsitellään optimointeja, joiden vertailuissa ei vaikuta satunnaisuus. Tällöin vertailu antaa aina saman tuloksen ja samoilla lähtöarvoilla alustettu optimointi etenee aina samalla tavalla. Tästä syystä ei ole ollut tarpeen toistaa samoja optimointeja kuten edellisessä luvussa. Painoarvojen konvergoimista joihinkin arvoihin on testattu suorittamalla optimointeja erilaisilla lähtöarvoilla ja tutkimalla päätyvätkö optimoinnit samoihin painoarvoihin.

Ei-satunnaisia optimointeja suoritettiin yhteensä kuusi. Kaikki kuusi optimointia optimoivat evaluointifunktiota *weight*10 ja käyttivät askeleen lyhentämistä optimoinnin edetessä. Askeleen lähtöarvoa ja sen lyhennyksen määrää vaihdeltiin. Kaikissa optimoinneissa käytettiin evaluointifunktion vahvuuden määräämiseksi vertailua funktioon *weight*10.4. Optimoinnin edetessä käytettiin kahta erilaista tapaa muuttaa painoarvoja: yksi kerrallaan tai kaikki kerrallaan. Nämä vastaavat intuitiivisesti n-ulotteisessa avaruudessa liikkumista yhden akselin suuntaan kerrallaan tai useamman akselin suuntaan yhdellä kertaa. Täten optimoinnissa käytetyt menetelmät voidaan periaatteessa samaistaa stokastiseen ja normaaliin gradienttimenetelmään [Lue84, sivut 214-215]. Taulukkoon 7 on kerätty tiedot suoritetuista kuudesta optimoinnista.

optimoinnin tulos	**lähtöaskel**	**lyhennys**	**painokerroinmuutos**	**aika**
*weight*10.10	10	0, 5	kaikki kerrallaan	2*min* 58*s*
*weight*10.11	20	0, 1	kaikki kerrallaan	34*min* 1*s*
*weight*10.12	15	0, 1	kaikki kerrallaan	25*min* 25*s*
*weight*10.15	10	0, 5	yksi kerrallaan	2*min* 54*s*
*weight*10.14	15	0, 1	yksi kerrallaan	30*min* 33*s*
*weight*10.13	20	0, 1	yksi kerrallaan	46*min* 27*s*

Taulukko 7: *Ei-satunnaisten optimointien lähtöarvot ja optimointiajat.*

Taulukosta 7 nähdään, että tehdyt testit jakautuvat kahteen eri ryhmään. Ensimmäisessä ryhmässä optimointi suoritettiin kolmella eri lähtöarvolla muuttaen kaikkia painokertoimia kerrallaan. Toisessa ryhmässä optimointi suoritettiin samoilla lähtöarvoilla muuttaen yhtä painokerrointa kerrallaan. Täten optimoinneista muodostui vertailtavia pareja. Ei-satunnaisiin optimointeihin kuluneet optimointiajat olivat paljon lyhyempiä kuin edellisen luvun optimoinneissa, koska vertailussa pelattiin vain kaksi peliä. Tämä mahdollisti optimointiaskelten määrän kasvattamisen suhteessa satunnaisiin optimointeihin.



Kuva 10: *Taulukon 7 optimointien tulokset matriisiesityksinä.*

Kuvassa 10 on visualisoitu ei-satunnaisten optimointitestien tulokset matriiseina. Tulokset on ryhmitelty niin, että ylärivillä ovat optimoinnit, joiden painoarvoja on muutettu kaikkia kerrallaan yhdessä optimointiaskeleessa. Alarivillä taas ovat niiden optimointien tulokset, joiden painoarvoja on muutettu yhtä kerrallaan. Ylärivi vaikuttaa kirjavammalta kuin alarivi. Tämä tarkoittaa, että ylärivin evaluointifunktioiden painoarvojen vaihtelu on suurempi kuin alarivin evaluointifunktioiden.

Matriisit on ryhmitelty myös niin, että sarakkeet muodostavat pareja. Päällekkäisiä matriiseja vastaavat optimoinnit ovat lähteneet samoista askelien pituuksista ja lyhennyksistä. Päällekkäisistä pareista vain ensimmäiset muistuttavat hieman toisiaan. Tämä voi johtua siitä, että ensimmäinen optimointi oli paljon lyhyempi kuin kaksi muuta. Optimointikierroksia kertyi ensimmäisessä optimoinnissa vain 20, kun toisessa ja kolmannessa kierroksia oli 200 ja 150. On mahdollista, että jos optimointia olisi jatkettu pidempään, olisivat ensimmäisenkin optimoinnin

tulokset edenneet kauemmas toisistaan. Toiset ja kolmannet painomatriisit eivät muistuta suuresti toisiaan, joten ne eivät ole konvergoineet kohti samoja painokertoimia.

Optimoinneissa lähtökohtana oli evaluointifunktio *weight*10, jonka painoarvoja lähdettiin muuttamaan. Taulukossa 8 on koottuna evaluointifunktion *weight*10 ja ei-satunnaisissa optimoinneissa saatujen evaluointifunktioiden suoriutumista vertailussa evaluointifunktiota *weight*10.4 vastaan. Ensimmäisellä rivillä on vertailun tulos pelikiekkoprosentteina, kun pelejä pelataan 2 kappaletta eikä peleissä vaikuta sattuma. Optimoinnissa käytetään täsmälleen samaa vertailua, joten ensimmäisen rivin vertailuprosentit osoittavat, että jokaisessa optimoinnissa on tapahtunut parannusta. Ennen optimointia oli pelikiekkoprosentti kaikilla 39, 0% ja optimoinnin jälkeen kaikki pääsivät yli 50% eli jokainen kehittyi paremmaksi kuin vertailufunktio *weight*10.4.

	10	**10.10**	**10.11**	**10.12**	**10.13**	**10.14**	**10.15**
2/ei-sat/kiekko	39, 0%	63, 3%	55, 5%	60, 9%	56, 3%	52, 3%	68, 0%
100/sat/kiekko	36, 2%	38, 8%	43, 1%	46, 3%	34, 4%	31, 0%	42, 2%
100/sat/voitto	16, 0%	15, 5%	27, 5%	35, 0%	11, 5%	2, 0%	23, 5%

Taulukko 8: *Ei-satunnaisten optimointien tulosten suoriutuminen vertailussa evaluointifunktioon weight*10.4.

Kuten edellisessä luvussa mainittiin, optimointialgoritmi ei koskaan muuta evaluointifunktiota huonompaan suuntaan. Sattuma ei vaikuta vertailuihin, joten vertailu ilmoittaa aina saman evaluointifunktion paremmaksi. Lisäksi koska kaikki vertailut tehdään evaluointifunktiota *weight*10.4 vastaan, tulkitaan parhaaksi evaluointifunktioksi se, joka pärjää parhaiten ennalta määrätyissä peleissä evaluointifunktiota *weight*10.4 vastaan. Vertailu evaluointifunktioon *weight*10.4 luo selkeän ja yksikäsitteisen mitan, jonka suhteen optimoitava evaluointifunktio joko paranee tai pysyy samana. Tästä syystä ei-satunnaisessa optimoinnissa ei voida päätyä tilanteeseen, jossa evaluointifunktio paranee paikallisesti koko ajan, mutta lopulta päätyy alkutilannetta huonompaan tilaan. Jokaisen muutetun evaluointifunktion pelikiekkoprosentti on parempi kuin edeltäjänsä.

Taulukon 8 ensimmäisellä rivillä on kuvattu optimoinnissa käytetty vertailu. Tämän mitan suhteen parannusta alkuperäiseen on tapahtunut. Taulukon 8 kahdella alimmalla rivillä on evaluointifunktioiden vertailuprosentteja suhteessa eva-

luointifunktioon *weight*10.4, kun vertailussa on pelattu 100 peliä satunnaisin siirroin. Keskimmäisellä rivillä on pelikiekkoprosentti ja alimmalla rivillä voittoprosentti. Satunnainen vertailu antaa paremman kuvan evaluointifunktion todellisesta vahvuudesta kuin optimoinnissa käytetty vertailu. Kun evaluointifunktiot ovat hyvin eri tasoisia, antaa pelikiekkoprosentti huonommalle hieman liian positiivisen vertailuarvon. Täten siis evaluointifunktioiden todellinen vertailuarvo suhteessa evaluointifunktioon *weight*10.4 on lähimpänä alimmaista riviä.

Vaikka optimoinnissa käytetyn vertailun asettaman mittarin mukaan optimointi tuotti selkeästi parempia evaluointifunktioita kuin alkuperäinen, eivät ne pärjänneet käytännössä alkuperäistä paremmin. Satunnaisia pelejä pelattaessa yksikään evaluointifunktio ei pääse evaluointifunktion *weight*10.4 tasolle ja osa suoriutuu peleistä jopa huonommin kuin alkuperäinen evaluointifunktio *weight*10. Tältä kannalta katsottuna optimointi ei ole parantanut evaluointifunktioita. Ongelmana optimoinnissa on ollut mitta, jonka suhteen evaluointifunktioita on parannettu. Se ei ole vastannut todellisuutta. Optimointi on tuottanut evaluointifunktioita, jotka pärjäävät ennalta määrätyssä vertailussa hyvin, mutta muissa vertailuissa jäävät suurin piirtein alkuperäisen tasolle.

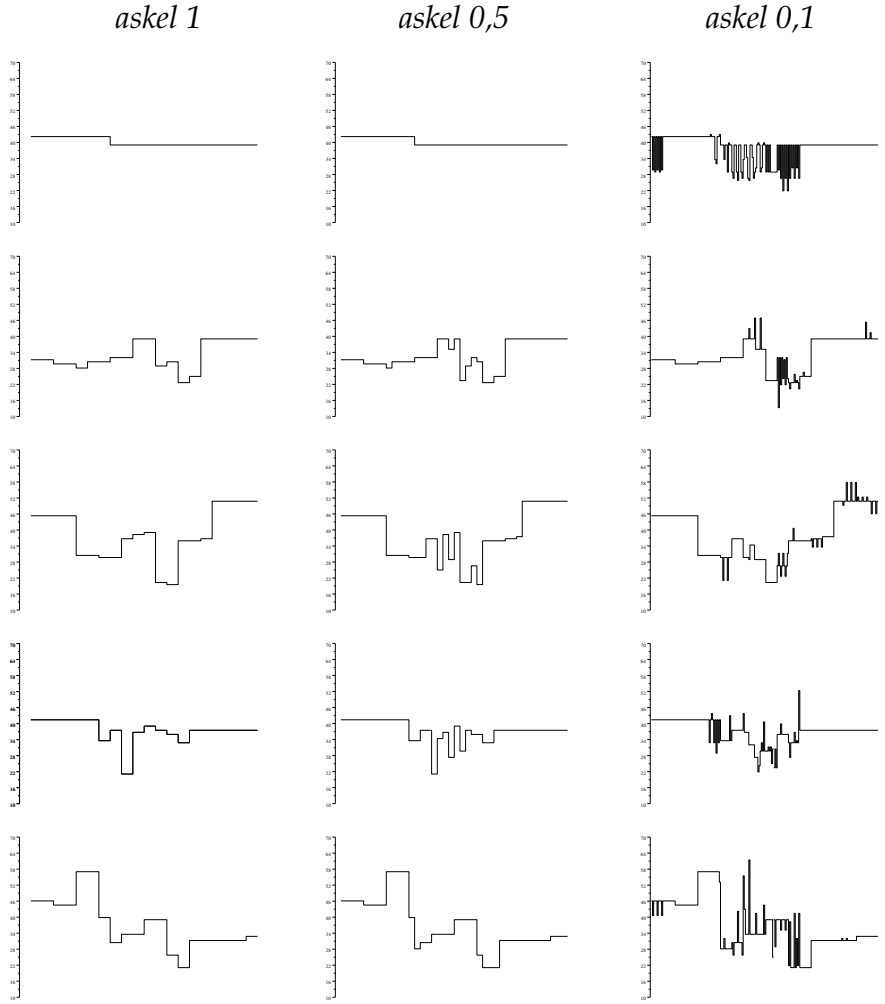
Kuvasta 10 nähdään, että ei-satunnaisessakin optimoinnissa painoarvot päätyvät hyvin eri suuntiin. Tämä ei voi johtua sattuman vaikutuksesta optimoinnin kulkuun kuten edellisessä luvussa, koska optimointi etenee suoraviivaisesti aina samojen askelten mukaan ja päätyy aina samasta alkutilasta samaan lopputilaan. Järkevin selitys painoarvojen satunnaisuudelle on optimoitavan funktion epäsäännöllisyys.

Optimoitava funktio on tämän luvun testien tapauksessa pelikiekkoprosentti kahden ennalta määrätyn pelin vertailussa evaluointifunktioon *weight*10.4. Optimoinnin kohdefunktio saa siis syötteenään evaluointifunktion ja palauttaa vertailun pelikiekkoprosentin. Tutkimalla tämän funktion käyttäytymistä erilaisilla evaluointifunktioilla, saadaan kuva siitä minkälainen optimoitava funktio on. Tämä on toteutettu tutkimalla miten kunkin painokertoimen muuttaminen evaluointifunktiosta *weight*10 vaikuttaa vertailuun.

Kohdefunktion laskenta etenee siten, että ensin lasketaan vertailuarvo evaluointifunktiolle. Tämän jälkeen yhtä painokerrointa muutetaan tietyn askelmäärän verran ja lasketaan näin saadun evaluointifunktion vertailuarvo. Vertailuarvot muodostavat kuvaajan, josta nähdään painokertoimen muutoksen vaikutus vertailuarvoon, kun muut painokertoimet pidetään alkuperäisinä. Mitä pienempi on

painokertoimen muutos, sitä tarkempi painokertoimen vaikutuksen kuvaaja on.

Kuvassa 11 on kuvattu 5:n ensimmäisen painokertoimen osalta niiden muutosten vaikutus alkuperäiseen evaluointifunktioon *weight*10. Liitteestä 3 löytyvät vastaavat kuvaajat kaikkien painokerrointen osalta. Painokertoimia on muutettu alkuperäisistä enintään 10 yksikköä molempiin suuntiin, joten painokertoimet ovat välillä $[-9, 11]$. Kunkin kuvaajan puolessa välissä on alkuperäinen tilanne. Vasemmalle mentäessä nähdään miten painokertoimen pienentäminen vaikuttaa pelikiekkoprosenttiin vertailussa evaluointifunktioon *weight*10.4. Oikealle mentäessä puolestaan nähdään miten painokertoimen suurentaminen vaikuttaa. Kunkin painokertoimen kuvaaja muodostaa yhden kolmen kuvan rivin. Vasemman puoleisissa kuvissa askeleen pituus on 1, keskimmäisissä kuvissa $0,5$ ja oikeanpuoleisissa kuvissa $0,1$. Näitä samoja askeleen pituuksia on käytetty myös optimoinneissa. Kuvaajat siis tarkentuvat kuva kuvalta oikealle mentäessä.



Kuva 11: *Optimoinnin kohdefunktion kuvausta, viisi ensimmäistä painokerrointa.*

Kuvaajat selventävät sitä, minkälainen ei-satunnaisen optimoinnin kohdefunktio on. Ne ovat funktion kymmenen akselin suuntaisten projektioiden pyöristyksiä tietyillä jakoväleillä. Kukin kuvaaja kuvaa yhden akselin suuntaista projektiota. Kuvaajat on muodostettu pisteestä $(1, 1, 1, 1, 1, 1, 1, 1, 1, 1)$ etäisyydelle 10 kunkin akselin suuntaan. Kuvasta 11 huomataan selvästi, että mitä pienempi jakoväli, sen epäsäännöllisemmäksi kuvaajat tulevat. Oikeanpuoleiset kuvaajat ovat lähimpänä todellista kuvaajaa, joten todellinen optimoitava funktio on todennäköisesti hyvin epäsäännöllinen. Tämä on todennäköinen syy sille, että optimoinnit eivät konvergoineet tiettyjä painoarvoja kohti. Hyvin pienet erot lähtötilanteissa voivat viedä optimoinnin täysin eri suuntaan ja täten päätyä täysin eri lopputulokseen. Jos optimoitava funktio on hyvin epäsäännöllinen, voi funktion käyttäytyminen optimipisteiden ympärillä olla lähes satunnaista. Tällöin funktio ei anna mitään tietoa optimipisteistä ja ainoaksi optimointimenetelmäksi jää pisteiden arpominen ja testaaminen [Gil81, sivut 93-94].

Optimoinnin toimivuus palautuu hyvän mittafunktion, eli luvun 4.1 funktion *power*, löytämiseen. On löydettävä mittafunktio, joka mittaa hyvin evaluointifunktion suoriutumista, ei sisällä satunnaisuutta ja käyttäytyy jokseenkin säännöllisesti. Tällöin optimointimenetelmiä voidaan järkevästi käyttää parempien evaluointifunktioiden etsimiseen.

5 Evaluointifunktion parantaminen geneettisellä ohjelmoinnilla

Tässä luvussa esitellään, miten evaluointifunktiota voi parantaa geneettisen ohjelmoinnin avulla. Luvun alussa esitellään geneettinen ohjelmointi. Tämän jälkeen esitellään miten tutkielmassa sitä sovelletaan evaluointifunktioiden parantamiseen ja lopuksi käydään pintapuolisesti läpi tehdyt testit ja niiden tulokset.

5.1 Geneettinen ohjelmointi

Geneettisessä ohjelmoinnissa (*genetic programming*) ideana on luoda tietokoneohjelmia luonnosta kopioidun evoluutiomallin periaattein [Nil98, sivut 59-60]. Prosessi on periaatteessa seuraavanlainen [Ban98, sivut 133-135]: luodaan ensin satunnainen joukko ohjelmia, testataan ne ja valitaan jonkin kriteerin mukaan parhaat. Näistä luodaan taas risteyttämällä uusi sukupolvi ohjelmia. Hyvin toteutettuna ohjelmat paranevat asetetun kriteerin suhteen sukupolvi sukupolvelta yleensä tiettyyn rajaan asti [Ban98, sivu 336]. Geneettisessä ohjelmoinnissa siis pyritään saamaan ohjelmien evoluution avulla halutunlainen ohjelma. Ohjelmat voivat olla muitakin kuin tietokoneohjelmia. Tässä tutkielmassa ohjelmat ovat evaluointifunktioita.

5.2 Geneettisen ohjelmoinnin toteutus

Tutkielmassa toteutettu geneettinen ohjelmointi on kuvattu yleisellä tasolla seuraavissa aliluvuissa. Tarkempi toteutuksen kuvaus löytyy tutkielman sovelluksen WWW-sivulta [ImpOthello].

5.2.1 Alkupopulaatio

Geneettinen ohjelmointi alkaa alkupopulaation luomisella [Nil98, sivu 62]. Alkupopulaatio luodaan yleensä satunnaisesti kuten tässäkin tutkielmassa. Alkupopulaation jäsenet ovat luvussa 3.3.1 esitellyn muotoisia evaluointifunktioita. Alkupopulaation luomisessa määrätään alifunktioiden maksimimäärä a sekä funktion maksimisyvyys s ja luodaan alkupopulaation koon mukainen määrä nämä rajoitteet täyttäviä evaluointifunktioita.

Alkupopulaatioon kuuluvan satunnaisen evaluointifunktion luominen tapahtuu siten, että arvotaan ensin funktion alifunktioiden määrä $(1 - a)$ ja alifunktioiden välissä olevat operaattorit. Tämän jälkeen sama toistetaan alifunktioille, kunnes maksimisyvyys s saavutetaan. Jos alifunktion koko on yksi tai maksimisyvyys on saavutettu, korvataan alifunktio satunnaisella perusfunktiolla. Tällä tavalla saadaan funktioita, joiden perusfunktioiden määrä vaihtelee välillä $1 - a^s$.

Perusfunktiot voivat olla joko pelilaudan ruutuja ($[i, j]$, $0 \leq i \leq 7$ ja $0 \leq j \leq 7$) tai positiivisia desimaalilukuja. Alkupopulaation luonnissa voi määrätä kuinka suuri osa perusfunktioista on pelilaudan ruutuja ja mikä on desimaalilukujen yläraja d. Satunnainen perusfunktio määritetään arpomalla ensin onko vakio pelilaudan ruutu vai desimaaliluku ja tämän jälkeen arpomalla joko satunnainen pelilaudan ruutu tai desimaaliluku väliltä $]0, d[$.

Populaation koko on eräs geneettisen ohjelmoinnin tärkeimmistä parametreista [Ban98, sivu 335]. Populaation koko pidetään yleensä samana koko geneettisen ohjelmoinnin ajan. Niin tehdään myös tässä tutkielmassa, joten alkupopulaation koko on hyvin tärkeä parametri. Populaation koko on tyypillisesti välillä 500 – 5000 [Ban98, sivu 311]. Positiivisia tuloksia on kuitenkin saatu $10 - 10^6$ yksilön populaatioilla, joten rajat eivät ole ehdottomia [Ban98, sivu 335].

5.2.2 Yksilöiden testaus

Yksilöiden testauksen tarkoituksena on saada selville yksilöiden keskinäinen paremmuusjärjestys. Tietoa tarvitaan, jotta voitaisi tehdä jonkinlaista luonnon valintaa. Populaatiosta karsitaan huonoja yksilöitä ja tuotetaan parempia tilalle. Tavoitteena on valita hyvät yksilöt, joista muodostetaan uusi sukupolvi.

Testauksen voi tehdä esimerkiksi siten, että yksilöt kilpailevat toisiaan vastaan jonkinlaisissa turnauksissa [Nil98, sivu 63]. Tällöin mitataan yksilöiden keskinäistä paremmuutta. Yksilöitä voidaan myös vertailla mittafunktion (*fitness function*) avulla, jolloin mitataan yksilöiden paremmuutta jonkin mittafunktion määräämän mitan suhteen. Mittafunktio antaa yksilölle arvon, joka kertoo sen hyvyyden [Ban98, sivut 130-132]. Molempia edellä mainittuja testaustapoja on käytetty tutkielmassa suoritetuissa kokeissa.

Testauksen pohjalta suoritetaan valinta. Valinta voidaan tehdä useilla eri periaatteilla vaikka yksilöiden testaus olisikin suoritettu samalla tavalla [Ban98, sivut 129-133]. Yksilöt voidaan pisteyttää tai asettaa paremmuusjärjestykseen turnaus-

ten tai mittafunktion perusteella. Pisteiden tai paremmuusjärjestyksen mukaan valinta voidaan tehdä valitsemalla suoraan parhaita yksilöitä tai perustaa valinta painotettuihin todennäköisyyksiin.

Tutkielmassa käytetyssä menetelmässä yksilön todennäköisyys tulla valituksi on suoraan verrannollinen tämän paikkaan keskinäisessä paremmuusjärjestyksessä. Järjestyksessä viimeinen yksilö saa painoarvon 1, toiseksi viimeinen painoarvon 2 ja niin edelleen. Populaation paras yksilö saa painoarvon, joka on sama kuin populaation koko. Yksilöt valitaan arpomalla ne painoarvojen suhteen painotetusta jakaumasta, jolloin hyvin pärjänneet yksilöt valitaan todennäköisimmin.

5.2.3 Geneettiset operaatiot

Uusi sukupolvi luodaan soveltamalla geneettisiä operaatioita hyviksi todettuihin yksilöihin. Geneettisiä operaatioita on kolme erilaista: jäljennös (*reproduction*), mutaatio (*mutation*) ja risteytys (*crossover*) [Ban98, sivu 122]. Kullakin näistä on oma biologinen vastaavuutensa.

Jäljennös tarkoittaa yksinkertaisesti yksilön kopioimista. Tällöin yksilö lisääntyy ikään kuin jakautumalla. Jäljennöksen jälkeen populaatiossa on kaksi identtistä yksilöä ja täten niiden geneettisen aineksen periytyvyyden todennäköisyys on korkeampi [Ban98, sivu 126]. Tässä tutkielmassa geneettinen ohjelmointi etenee sukupolvi kerrallaan siten, että uusi sukupolvi luodaan vanhasta ja vanha tuhotaan. Tällöin jäljennös toteutetaan niin, että kopioidaan jäljennettävä yksilö suoraan uuteen sukupolveen [Ban98, sivu 130]. Tällöin esimerkiksi satunnainen huippuyksilö voi säilyä alkuperäisessä muodossaan sukupolvesta toiseen.

Mutaatio tarkoittaa satunnaisen muutoksen tekemistä yksilöön [Ban98, sivut 125-126]. Mutaation biologinen vastaavuus on luonnossa tapahtuva mutaatio, joka positiivisessa tapauksessa edistää evoluutiota. Mutaatiot tuovat vaihtuvuutta populaation geneettiseen ainekseen. Mutaation voi toteuttaa hyvin monella eri tavalla [Ban98, sivut 240-242]. Tässä tutkielmassa mutaatiot on toteutettu siten, että arvotaan sovelletaanko mutaatiota perusfunktioon vai operaattoriin. Operaattorin mutaatio tehdään arpomalla jokin funktion operaattori ja korvaamalla se satunnaisella operaattorilla (taulukko 9). Perusfunktion mutaatio tehdään arpomalla jokin perusfunktio ja korvaamalla se satunnaisella perusfunktiolla (taulukko 9). Satunnaiset perusfunktiot määrätään samalla tavalla kuin alkupopulaation luonnissa.

operaattorin mutaatio	$1 + [1,2] - [3,2] \Rightarrow 1 + [1,2] * [3,2]$
perusfunktion mutaatio	$1 + [1,2] - [3,2] \Rightarrow 1 + [1,2] - 4$

Taulukko 9: *Esimerkit mutaatioista.*

Risteytys on kaikkein käytetyin geneettinen operaatio [Nil98, sivu 64]. Se saa motivaationsa luonnon pariutumisesta, jossa kahdesta eri yksilöstä tuotetaan populaatioon uusia yksilöitä. Uusien yksilöiden geeniperimä on molempien vanhempien geeniperimien risteytys. Risteytyksessä etuna on, että molempien vanhempien hyvät ominaisuudet voivat periytyä uudelle yksilölle. Täten saadaan aikaan mahdollisesti parempi yksilö kuin kumpikaan vanhemmista ja evoluutio etenee.

Geneettisessä ohjelmoinnissa risteytys voidaan toteuttaa monin eri tavoin [Ban98, sivut 240-241]. Tässä tutkielmassa käytetään suoraviivaista tapaa, jota sovelletaan usein puumuodossa esitettäville ohjelmille. Menetelmässä kahdesta yksilöstä risteytetään kaksi jälkeläistä. Ensin valitaan satunnaiset alifunktiot vanhempien evaluointifunktioista. Jälkeläiset saadaan vaihtamalla valitut vanhempien alifunktiot keskenään, jolloin molemmat jälkeläiset koostuvat toisen vanhempansa alku- ja loppuosasta ja toisen vanhempansa keskiosasta (taulukko 10). Tämä risteytystapa vastaa sitä, että kahden puuna esitettävän ohjelman kaksi satunnaista alipuuta vaihdetaan keskenään [Mic96, sivu 286].

risteytys (isä)	$(1 + [3,2])+\mathbf{(}(4 + [6,7]) + 1 + ([5,3] + [3,5])\mathbf{)}+[4,3]$
risteytys (äiti)	$(1 + [2,4]+\mathbf{(}[3,3] + 6\mathbf{)}) + 5$
risteytys (lapsi1)	$(1 + [3,2])+\mathbf{(}[3,3] + 6\mathbf{)}+[4,3]$
risteytys (lapsi2)	$(1 + [2,4]+\mathbf{(}(4 + [6,7]) + 1 + ([5,3] + [3,5])\mathbf{)}) + 5$

Taulukko 10: *Esimerkki risteytyksestä.*

Yleensä geneettisessä ohjelmoinnissa määrätään missä suhteessa eri geneettisiä operaatioita käytetään [Ban98, sivu 337]. Risteytysten määrä on usein suuri ja mutaatioiden pieni, kuten luonnossa. Jäljennöksiä käytetään ongelmasta riippuen. Tutkielmassa geneettiset operaatiot on toteutettu siten, että aluksi määrätään jäljennös-, mutaatio- ja risteytysprosentit. Tämän jälkeen joka sukupolvi testataan. Tulosten perusteella kopioidaan vanhan sukupolven yksilöistä jäljennösprosentin verran parhaita yksilöitä uuteen sukupolveen. Mutaatioprosentin määräämälle määrälle satunnaisia yksilöitä suoritetaan mutaatiot, joiden tulokset lisätään uuteen sukupolveen. Viimeiseksi suoritetaan risteytykset. Luvussa 5.2.2 kuva-

tulla tavalla valitaan kaksi evaluointifunktiota ja risteytetään ne. Tätä toistetaan kunnes ollaan saatu risteytysprosentin verran uusia yksilöitä. Lopuksi nämä lisätään uuteen sukupolveen.

5.3 Suoritettuja geneettisen ohjelmoinnin testejä

Tutkielman puitteissa on suoritettu kolme geneettisen ohjelmoinnin testiä. Tässä luvussa esitellään ne suoritusjärjestyksessä. Kaikkien testien raportit ja evaluointifunktioiden sukupolvet löytyvät tutkielman tulosten WWW-sivulta [Sii02]. Geneettinen ohjelmointi on luonteeltaan hyvin aikaa vievää [Ban98, sivut 311-312] ja siksi tutkielman puitteissa on ollut mahdollista suorittaa vain nämä testit.

5.3.1 Ensimmäinen testi

Ensimmäisessä testissä populaation jäsenet kilpailivat toisiaan vastaan. Kilpailu toteutettiin turnauksena, jossa jokainen populaation evaluointifunktio pelasi jokaista muuta evaluointifunktiota vastaan. Pelejä pelattiin kaksi kappaletta (mustilla ja valkoisilla) ennalta määrätyin siirroin ja pelaajat saivat pisteitä pelikiekkoprosenttien mukaan. Pelaajan vahvuus turnauksessa oli kaikkien pelattujen pelien pelikiekkoprosenttien summa.

Yhden sukupolven käsittelyyn kuluva aika oli suurin piirtein sama kuin turnaukseen kuluva aika, koska turnaus muodosti selkeästi aikaa vievimmän osan sukupolven käsittelystä. Turnaukseen kuluva aika oli osallistujien lukumäärän suhteen neliöllinen. Tämä rajoitti suuresti populaation kokoa. Ensimmäisessä testissä käytettiin populaatiolle muuttumatonta kokoa 100.

Alkupopulaation jäsenet muodostettiin satunnaisesti siten, että niiden maksimipituus oli 5 ja maksimisyvyys 4. Evaluointifunktioista oli 50% satunnaisia pelilaudan ruutuja ja loput satunnaisia vakioita väliltä $]0, 5[$. Jäljennösprosentiksi valittiin 10% ja risteytysprosentiksi 90%. Mutaatioita ei tehty lainkaan. Koska prosenttien summa oli 100%, pysyi populaation koko samana sukupolvesta toiseen. (Katso ensimmäisen testin yhteenveto taulukosta 11.)

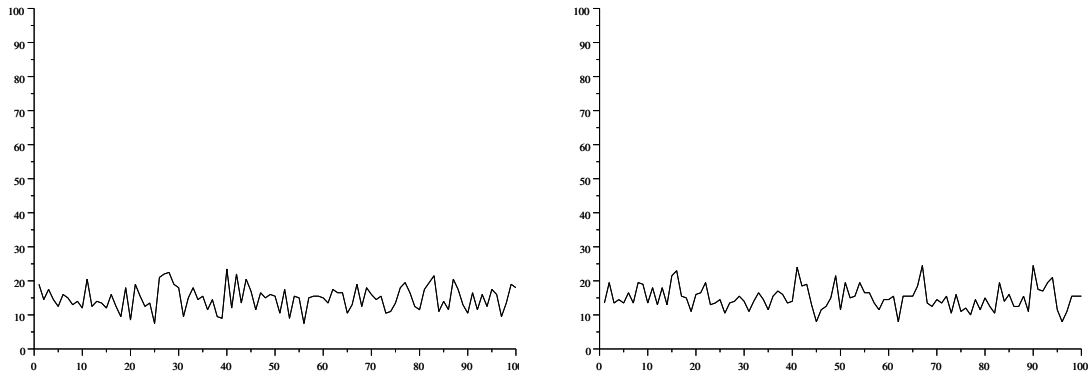
Yhden sukupolven käsittelyyn meni yhdestä tunnista kolmeen tuntiin. Kaiken kaikkiaan geneettistä ohjelmointia suoritettiin 100 sukupolven verran ja tähän meni hieman yli viikko laskenta-aikaa. Kuvassa 12 on ensimmäisen testin kehityskäyrät kahdella eri mittarilla. Käyrät on muodostettu vertaamalla jokaisen su-

alkupopulaatio	max pituus 5, max syvyys 4, perusfunktioista 50% vakioita]0, 5[ja loput pelilaudan ruutuja
mittafunktio	keskinäinen turnaus
populaation koko	100
päivitysprosentit	jäljennös 10 %, risteytys 90 %, mutaatio 0%
sukupolvien määrä	100
laskenta-aika	hieman yli viikko

Taulukko 11: *Ensimmäisen testin yhteenveto.*

kupolven (0 – 100, vaaka-akseli) turnauksen tuloksen mukaan parasta evaluointifunktiota kiinteisiin evaluointifunktioihin *weight*10.4 (vasen kuvaaja) ja *weight*10.5 (oikea kuvaaja).

Vertailut suoritettiin peluuttamalla evaluointifunktioita vastakkain 100 kertaa satunnaisin siirroin ja laskemalla vertailtavien evaluointifunktioiden voittoprosentit. Käyrät ovat siis sukupolvien parhaiden yksilöiden voittoprosentteja suhteessa vertailtaviin evaluointifunktioihin. Vertailtavat evaluointifunktiot *weigth*10.4 ja *weight*10.5 on valittu sillä perusteella, että ne menestyivät parhaiten luvun 4 optimoinneissa. Satunnainen pelityyli on valittu siksi, että se antaa paremman kuvan evaluointifunktion todellisesta tasosta kuin ennalta määrätyt evaluointifunktioiden väliset pelit.



vertailufunktiona *weight*10.4 vertailufunktiona *weight*10.5

Kuva 12: *Kaksi ensimmäisen testin kehityskäyrää (voittoprosentti).*

Pelien satunnaisuus tuo mukanaan kuvaajiinkin satunnaisuutta. Jos käyrät piirrettäisiin uudelleen, olisivat ne eri näköisiä. Satunnaisuuden vaikutus on kuiten-

kin sen verran pieni, että käyrät pysyvät tiettyjen rajojen sisällä. Parempi kehityskäyrä saataisiin yhdistämällä useita satunnaisia kehityskäyriä.

Kuvan 12 käyrät eivät nouse havaittavasti loppua kohti. Tämä tarkoittaa, että suhteessa evaluointifunktioihin *weigth*10.4 ja *weight*10.5 sukupolvien parhaissa evaluointifunktioissa ei tapahtunut juuri yhtään parannusta. Syitä tähän voi olla monia. Geneettinen ohjelmointi on saattanut parantaa evaluointifunktioita jossain suhteessa, mutta ei suhteessa evaluointifunktioihin *weigth*10.4 ja *weight*10.5. Tämä ei kuitenkaan olisi tavoiteltavaa. Tavoitteena on kehittää yleisesti hyviä evaluointifunktioita, jolloin niiden pitäisi pärjätä hyvin myös suhteessa evaluointifunktioihin *weigth*10.4 ja *weight*10.5.

Huonot geneettisen ohjelmoinnin parametrien valinnat voivat olla syynä ensimmäisen testin tuloksiin. Voi olla, että alkupopulaatio ei ollut tarpeeksi monipuolinen tai että populaation koko ei ollut tarpeeksi suuri. Tehokkuutta geneettisen ohjelmoinnin etenemiseen olisivat voineet tuoda mutaatiot tai suurempi jäljennösprosentti.

Eräs mahdollinen syy ensimmäisen testin tehottomuuteen ovat epämielekkäät evaluointifunktiot. Nämä ovat sellaisia, jotka tuottavat epämielekkäitä evaluointeja nollalla jakamisen takia. Koska evaluointifunktiot muodostetaan satunnaisesti, on mahdollista, että arvojen evaluoinnissa tapahtuu joissain pelitilanteissa nollalla jakamista. Epämielekkäät evaluoinnit vääristävät pelin kulkua ja täten vertailujen tuloksia.

Tarkastelemalla ensimmäisen geneettisen ohjelmoinnin testin etenemistä (katso [Sii02]) huomataan, että suurin osa (63%) sukupolvien parhaista yksilöistä on edellisen sukupolven parhaita yksilöitä, vaikka 90% sukupolven yksilöistä on risteytyksen tuotteita. Lisäksi vain kahden sukupolven kohdalla on niin, että paras yksilö ei ole joku edellisestä sukupolvesta säilytetyistä yksilöistä. Risteytykset eivät siis jostain syystä tuottaneet yksilöitä, jotka pärjäisivät seuraavassa sukupolvessa. Tällöin ei ole yllättävää, että kehitystä ei juurikaan tapahtunut.

5.3.2 Toinen testi

Toisessa testissä pyrittiin korjaamaan ensimmäisessä testissä havaittuja puutteita. Jakamisoperaatiota muutettiin siten, että nollalla jakamisen tulokseksi tuli aina nolla. Tällöin pelitilanteiden evaluoinnit tuottivat aina jonkin reaaliluvun. Toisessa testissä luovuttiin myös evaluointifunktioiden vertailusta toisiinsa. Sukupol-

ven yksilöiden hyvyys määriteltiin vertaamalla niitä evaluointifunktioon *weight*10.4. Vertailussa pelattiin 10 peliä satunnaisin siirroin, jotta sukupolven käsittelyyn kuluva aika ei kasvaisi liian suureksi. Pienen pelien määrän takia vertailun tulos laskettiin pelikiekkoprosentin mukaan voittoprosentin sijasta.

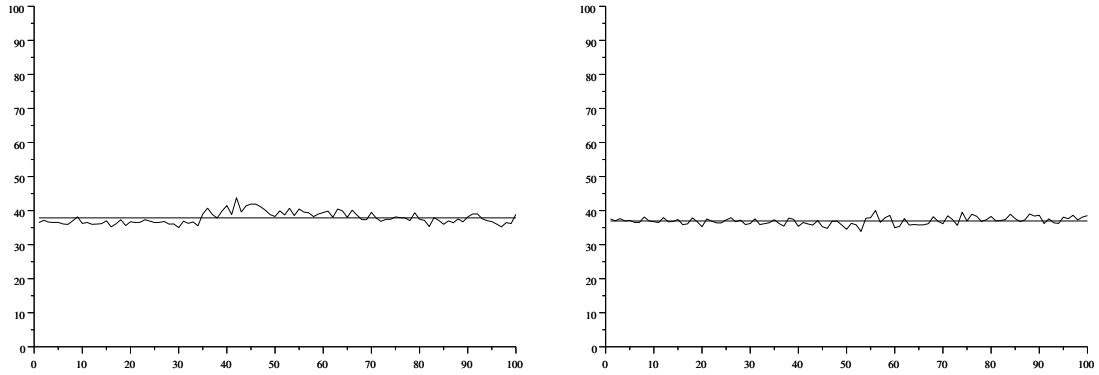
Toisen testin alkupopulaatioon otettiin 500 evaluointifunktiota. Alkupopulaation jäsenten maksimipituus oli 7 eli ne olivat pidempiä kuin ensimmäisessä testissä. Muut alkupopulaatioon liittyvät parametrit olivat toisessa testissä samat kuin ensimmäisessä. Toisessa testissä käytettiin jäljennösprosenttia 10%, mutaatioprosenttia 10% ja risteytysprosenttia 80%. Täten populaation koko pysyi samana sukupolvesta toiseen. (Katso toisen testin yhteenveto taulukosta 12.)

alkupopulaatio	max pituus 7, max syvyys 4, perusfunktioista 50% vakioita $]0, 5[$ ja loput pelilaudan ruutuja
mittafunktio	10 satunnaista peliä evaluointifunktiota *weight*10.4 vastaan
populaation koko	500
päivitysprosentit	jäljennös 10 %, risteytys 80 %, mutaatio 10%
sukupolvien määrä	100
laskenta-aika	hieman yli 8 vuorokautta

Taulukko 12: *Toisen testin yhteenveto.*

Hyvien yksilöiden valintaan käytetty vertailu ja populaation koko oli valittu siten, että yhden sukupolven käsittelyyn kuluva aika ei kasvaisi liian suureksi. Ensimmäisen sukupolven käsittelyyn meni aikaa noin 40 minuuttia. Sukupolvien käsittelyyn kuluva aika kasvoi kuitenkin sukupolvi sukupolvelta, koska evaluointifunktioiden keskimääräinen koko kasvoi. Tämä tarkoitti, että pidemmät evaluointifunktiot pärjäsivät lyhyitä paremmin. Geneettistä ohjelmointia jatkettiin 100 sukupolven ajan ja viimeisen sukupolven käsittelyyn kului aikaa yli 4 tuntia. Kaiken kaikkiaan toisen testin ajaminen vei yli 8 vuorokautta laskenta-aikaa.

Kuvassa 13 on toisen testin kehityskäyrät vertailussa funktioihin *weight*10.4 ja *weight*10.5. Käyriä määritettäessä otettiin kymmenen parasta evaluointifunktiota joka sukupolvesta. Näin tehtiin koska toisessa testissä sattuma vaikutti vertailuun melko paljon eikä vertailun mukaan paras sukupolven yksilöistä ollut aina välttämättä paras. Jokaista kymmentä evaluointifunktiota peluutettiin kahta vertailuevaluointifunktiota vastaan pelaamalla 30 peliä satunnaisilla siirroilla. Tulos määrättiin laskemalla pelikiekkoprosenttien keskiarvo. Tällä tavalla saatiin



vertailufunktiona *weight*10.4 vertailufunktiona *weight*10.5

Kuva 13: *Kaksi toisen testin kehityskäyrää ja keskiarvot (pelikiekkoprosentti).*

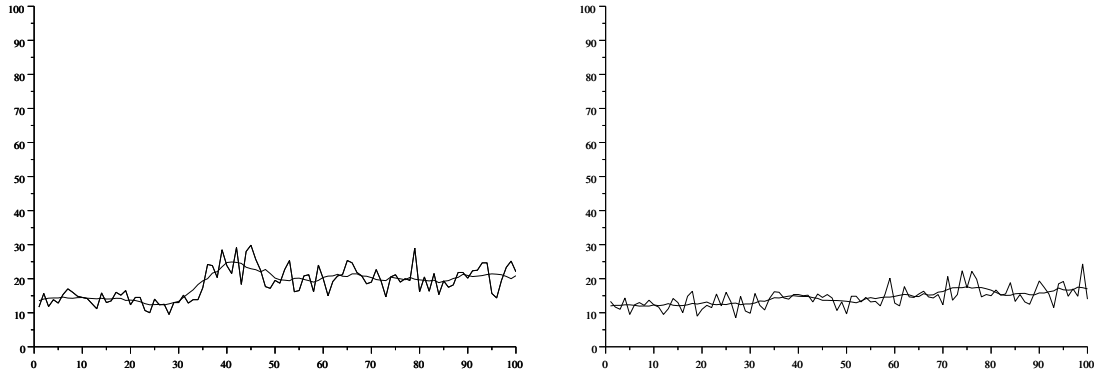
satunnaisuuden vaikutusta vähennettyä. Kuvan käyriin piirrettiin myös niiden keskiarvoa vastaava suora käyrien muutosten havaitsemisen helpottamiseksi.

Kuvasta 13 nähdään, että evaluointifunktioiden käyrät ovat melko tasaisia. Vasemman puoleinen käyrä käy keskiarvoviivan yläpuolella noin 40. sukupolven kohdalla, mutta palaa takaisin keskiarvon alapuolelle, kun lähestytään sukupolvea 100. Oikean puoleinen käyrä puolestaan hivuttautuu keskiarvoviivan yläpuolelle, kun sukupolvet lähestyvät 100:a.

Kehityskäyrien tasaisuus selittyy osaksi pelikiekkoprosentin tasaisuudella. Ensimmäiset sukupolvet, joissa on satunnaisia evaluointifunktioita, saavuttavat lähes 40% tason. Jos pelikiekkoprosentti olisi yli 50%, olisi evaluointifunktio jo periaatteessa parempi kuin vertailtavansa. Täten matka lähes 40%:sta 50%:iin on melko pitkä vaikka kuvaajassa se ei ole kovin suuri harppaus. Tilannetta selventää kuva 14, jossa on muodostettu toisen testin kehityskäyrät samalla tavalla kuin kuvassa 13, mutta vertailun tulos on laskettu voittoprosentteina.

Kuvan 14 käyrissä on enemmän satunnaista vaihtelua kuin kuvan 13 käyrissä. Tämän takia kuvan 14 käyriin on piirretty myös regressiokäyrä, joka poistaa satunnaisuutta. Regressiokäyrä on muodostettu laskemalla jokaisessa pisteessä keskiarvo viisisäteisessä ympäristössä eli laskemalla kymmenen lähimmän pisteen keskiarvo. Tällöin satunnaisten vaihtelujen vaikutus käyrään tasaantuu. Regressiokäyrä saattaa hieman vääristyä päissään, sillä päissä ei välttämättä ole kymmentä lähintä pistettä.

Kuvasta 14 nähdään, että toisessa testissä tapahtuu kehitystä evaluointifunktiois-



vertailufunktiona *weight*10.4 vertailufunktiona *weight*10.5

Kuva 14: *Kaksi toisen testin kehityskäyrää ja niiden regressiokäyrät (voittoprosentti).*

sa. Alussa molemmat evaluointifunktioiden voittoprosentit olivat hieman yli 10% ja lopussa ne olivat hieman yli ja alle 20%. Vertailufunktion *weight*10.4 tapauksessa käyrä nousee 40. sukupolven kohdalla ja palaa myöhemmin alaspäin, kuten myös kuvassa 13. Täten voimme olettaa, että evaluointifunktion *weight*10.4 suhteen geneettisen ohjelmoinnin huippu saavutettiin suunnilleen sukupolven 40 kohdalla. Vertailussa evaluointifunktioon *weight*10.5 parannusta tapahtui hieman koko ajan.

5.3.3 Kolmas testi

Kolmannessa testissä haluttiin saada populaation kokoa suuremmaksi kuin edellisissä testeissä. Tämän mahdollistamiseksi täytyi vertailuun kuluvaa aikaa pienentää. Aika saatiin pienenemään käyttämällä vertailua, jossa pelattiin ennalta määrättyjä pelejä. Tällöin yhtä vertailua kohti riitti pelata kaksi peliä. Koska pelejä pelattiin niin vähän, laskettiin vertailuarvot pelikiekkoprosentteina.

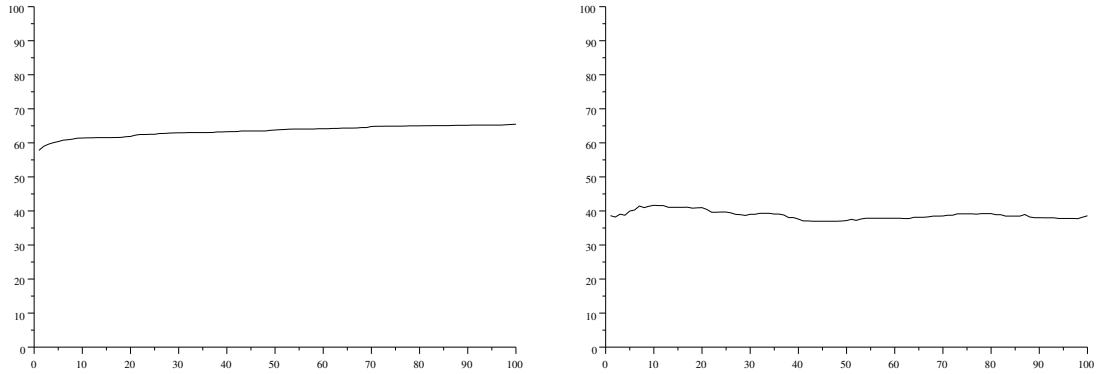
Kolmannessa testissä populaation koko oli 2000 yksilöä eli neljä kertaa toisen testin populaation koko. Lisäksi mutaatioprosenttia nostettiin, koska tätä keinoa voi käyttää, jos kehitystä ei tapahdu riittävästi [Ban98, sivu 337]. Kolmannessa testissä käytettiin jäljennösprosenttia 10%, mutaatioprosenttia 30% ja risteytysprosenttia 60%. Muuten parametrit olivat kuten toisessa testissä. (Katso kolmannen testin yhteenveto taulukosta 13.)

Kolmannessa testissä 100 sukupolven laskemiseen kului aikaa hieman yli 40 tuntia. Vaikka siinä pelattiin $\frac{4}{5}$ toisen testin pelien määrästä, aikaa meni vain noin

alkupopulaatio	max pituus 5, max syvyys 4, perusfunktioista 50% vakioita $]0, 5[$ ja loput pelilaudan ruutuja
mittafunktio	2 ennalta määrättyä peliä funktiota *weight*10.4 vastaan
populaation koko	2000
päivitysprosentit	jäljennös 10 %, risteytys 60 %, mutaatio 30%
sukupolvien määrä	100
laskenta-aika	40 tuntia

Taulukko 13: *Kolmannen testin yhteenveto.*

$\frac{1}{5}$ toiseen testiin kuluneesta ajasta. Kolmas testi oli siis huomattavan nopea verrattuna edellisiin. Yhden sukupolven käsittelyyn kuluva aika oli pienimmillään noin 11 minuuttia ja suurimmillaankin vain noin 41 minuuttia [Sii02]. Kolmannen testin nopeus johtui pääasiassa siitä, että suuremmat evaluointifunktiot eivät pärjänneet pienempiä paremmin ja täten evaluointifunktioiden koko ei suuresti kasvanut geneettisen ohjelmoinnin edetessä.



vertailufunktiona *weight*10.4 vertailufunktiona *weight*10.5

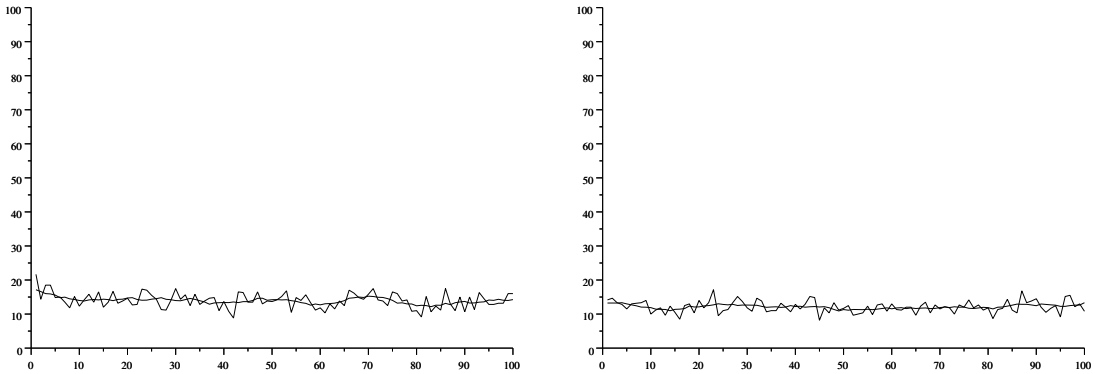
Kuva 15: *Kaksi kolmannen testin kehityskäyrää (pelikiekkoprosentit).*

Kehityskäyrät kolmannesta testistä ovat kuvassa 15. Vertailut on jälleen tehty suhteessa evaluointifunktioihin *weight*10.4 ja *weight*10.5. Käyrät on muodostettu pelaamalla kaksi ennalta määrättyä peliä jokaisen sukupolven 50 parasta yksilöä vastaan ja laskemalla pelikiekkoprosenttien keskiarvo.

Vasemman puoleinen käyrä tuotettiin samalla vertailulla kuin mitä kolmannessa testissä käytettiin mittaamaan evaluointifunktion hyvyyttä. Koska jäljennösprosentti oli 10%, jokaisen sukupolven 200 parasta yksilöä kopioitiin suoraan seu-

raavaan sukupolveen. Täten oli odotettua, että vasemman puoleinen käyrä on kasvava. Yllättävää kuitenkin on, että käyrä lähtee niin korkealta. Jo ensimmäisessä, satunnaisesti muodostetussa, sukupolvessa oli niin paljon hyviä evaluointifunktioita, että 50 parhaan pelikiekkoprosentin keskiarvo oli yli 57%. Itse asiassa 100. sukupolven paras yksilö on sama kuin alkupopulaation paras yksilö. Tämän huippuyksilön pelikiekkoprosentti oli yli 75% ja se pysyi parhaana koko geneettisen ohjelmoinnin ajan (katso [Sii02]). Kolmas testi ei siis onnistunut kehittämään parempaa yksilöä kuin sattumalta oli syntynyt, mutta se onnistui kehittämään koko populaatiota hieman paremmaksi. Populaation kehittyminen nähdään kuvan 15 vasemman puoleisesta käyrästä.

Kuvan 15 käyrä, jossa vertailufunktiona on *weight*10.5, ei vaikuta kasvavalta vaan aaltoilee edes takaisin. Täten kolmas testi ei kehittänyt suuremmin evaluointifunktioita ennalta määrättyjen pelien osalta vertailussa funktioon *weight*10.5. Tämä ei ole kovin yllättävää, sillä kuten luvussa 4 nähtiin, on ennalta määrättyjen pelien pelaaminen melko huono mitta evaluointifunktioiden hyvyydelle.



vertailufunktiona *weight*10.4 vertailufunktiona *weight*10.5

Kuva 16: *Kaksi kolmannen testin kehityskäyrää (voittoprosentit) ja näiden regressionkäyrät.*

Evaluointifunktioita on testattu myös niin, että ne pelaavat satunnaisilla siirroilla eri vastustajia vastaan. Kuvassa 16 ovat näin saadut kolmannen testin kehityskäyrät. Käyrät on muodostettu samalla tavalla kuin kuvan 14 kehityskäyrät. Jokaisen sukupolven kymmentä parasta evaluointifunktiota on peluutettu 30 kertaa vertailuevaluointifunktiota vastaan ja laskettu voittoprosenttien keskiarvo. Tämä mitta antaa kuvaa siitä, miten evaluointifunktiot pärjäävät todellisuudessa.

Kuvan 16 käyrät pysyttelevät sukupolvesta toiseen suurin piirtein samalla tasol-

la, joten sukupolvien parhaiden yksilöiden voittojen määrä ei lisäänny geneettisen ohjelmoinnin edetessä. Kuvan 15 vasemman puoleinen käyrä kuitenkin osoittaa, että jonkinlaista kehitystä on tapahtunut. Näyttäisi siltä, että evaluointifunktiot ovat kehittyneet suuntaan, joka parantaa niitä ei-satunnaisessa vertailussa evaluointifunktioon *weight*10.4. Tämä suunta on kuitenkin siinä mielessä väärä, että se ei paranna evaluointifunktioita yleisellä tasolla.

Kolmannessa testissä havaittiin, että ennalta määrättyjen pelien pelaaminen vertailuissa ei vienyt geneettistä ohjelmointia toivottuun suuntaan. Satunnaisten pelien pelaaminen näytti tuottavan parempia tuloksia. Ne vaativat kuitenkin useampia pelejä, jotta satunnaisuuden vaikutus tasaantuisi. Toteutetulla geneettisen ohjelmoinnin ympäristöllä satunnaiset vertailut ovat kuitenkin hyvin aikaa vieviä. Tämän takia populaation kokoa ja vertailussa pelattavien pelien määrää ei olisi voitu kovin paljon enää kasvattaa.

6 Yhteenveto

Tässä tutkielmassa on käsitelty kahden pelaajan nollasummapelien pelaamista ja erityisesti sitä miten tietokone oppii pelaamaan tällaisia pelejä. Esimerkkipelinä on ollut Othello ja oppiminen on tutkielman kannalta ollut evaluointifunktion kehittymistä. Evaluointifunktioiden parantamiseen on esitelty kaksi menetelmää ja näitä on käytännössä testattu. Tässä luvussa käydään ensin läpi suoritettujen testien tuloksia, analysoidaan niitä ja tarkastellaan miten menetelmät toimivat. Tämän jälkeen luodaan vielä yleiskatsaus peleihin ja koneoppimiseen tutkielman näkökulmasta.

6.1 Tulosten analysointia

Tutkielman kokeellisen osuuden muodostivat käytännön testit, joissa testattiin optimoinnin ja geneettisen ohjelmoinnin toimivuutta evaluointifunktioiden parantamisessa. Seuraavassa kahdessa aliluvussa analysoidaan tehtyjen testien tuloksia. Kolmannessa aliluvussa pyritään selvittämään kuinka hyvä pelitaso tutkielman testeissä on saavutettu.

6.1.1 Optimointi

Optimoinnissa tietyn tyyppisen evaluointifunktion vakioiden arvot pyrittiin saamaan mahdollisimman hyviksi. Tutkielmassa optimoitiin vain painoarvoihin perustuvia Othellon evaluointifunktioita. Testeissä pyrittiin siis löytämään mahdollisimman hyviä painoarvoja pelilaudan ruuduille. Testejä suoritettiin kahta eri tyyppiä: satunnaisia ja ei-satunnaisia. Satunnaisissa testeissä optimointi suoritettiin vertailun perusteella, jossa pelattiin 100 peliä satunnaisin siirroin ja laskettiin pelaajien voittoprosentit. Ei-satunnaisissa testeissä käytetyissä vertailuissa puolestaan pelattiin 2 ennalta määrättyä peliä ja laskettiin pelaajien pelikiekkoprosentit.

Satunnaisilla optimoinneilla saatiin aikaan vaihtelevasti jonkin verran kehitystä. Kehitys kuitenkin törmäsi melko nopeasti ylärajaansa. Ensimmäisellä optimointikierroksella saatiin positiivisia tuloksia. Kun tämän optimoinnin tuloksena saatua evaluointifunktiota yritettiin parantaa, ei uusi optimointikierros enää tuottanut parempia evaluointifunktioita. Eräs oleellinen tekijä kehityksen pysähtymiselle tuntui olevan vertailuissa vaikuttava satunnaisuus. Tietyn rajan yli mentäes-

sä satunnaisuuden vaikutus kasvoi niin suureksi, että optimointi ei hakeutunut enää paikallisiin huippuihin.

Ei-satunnaisilla optimoinneilla saatiin aikaan kehitystä mitan suhteen, joka määräytyi käytetyn vertailun mukaan. Jokainen optimoinnin tuloksena saatu evaluointifunktio pärjäsi alkuperäistä paremmin tätä mittaa käyttävässä vertailussa. Ongelmana optimoinnissa oli, että mitta ei vastannut todellisuutta. Vaikka evaluointifunktiot kehittyivät tietyssä vertailussa, eivät ne pärjänneet muissa vertailuissa alkuperäistä paremmin.

Eräs optimointitestien tavoite oli testata konvergoivatko painoarvot tiettyihin lukemiin. Vastaus tähän kysymykseen oli kielteinen. Ainakin käytetyillä menetelmillä evaluointifunktioiden painoarvot päätyivät aina eri lukemiin. Selitys tähän löytynee optimointien kohdefunktioista. Ei-satunnaisen optimoinnin kohdefunktio vaikuttaa hyvin epäsäännölliseltä (luku 4.3.4 ja liite 3). Myös satunnaisten optimointien kohdefunktiot voi olettaa hyvin epäsäännöllisiksi, koska niihin vaikuttaa sattuma. Liian epäsäännöllisen funktion optimointi ei ole mielekästä [Gil81, sivu 93]. Optimoinnin toimiminen evaluointifunktioiden kehittämismenetelmänä palautuukin tutkielman osalta säännöllisen optimoitavan funktion etsimiseen. Tämä puolestaan palautuu oikeellisen ja säännöllisen funktion *power* löytämiseen eli sopivan evaluointifunktioiden paremmuuden mittafunktion etsimiseen.

6.1.2 Geneettinen ohjelmointi

Geneettisessä ohjelmoinnissa tavoitteena oli evaluointifunktion parantaminen sukupolvi sukupolvelta. Satunnaisesti muodostetun alkupopulaation pohjalta luotiin uusia sukupolvia suosien parempia yksilöitä. Tutkielman puitteissa suoritettiin kolme testiä, joissa evaluointifunktioita pyrittiin kehittämään geneettisellä ohjelmoinnilla.

Ensimmäisessä testissä evaluointifunktiot kilpailivat keskenään ja parhaat yksilöt kopioitiin suoraan seuraavaan sukupolveen. Loput seuraavan sukupolven jäsenistä saatiin risteyttämällä edellisen sukupolven parhaat yksilöt. Mutaatioita ei käytetty lainkaan. Ensimmäinen testi ei tuottanut havaittavaa kehitystä evaluointifunktioissa. Todennäköisin syy tälle oli populaation pieni koko (100 yksilöä) ja evaluointifunktioissa tapahtunut nollalla jakaminen.

Toisessa testissä evaluointifunktioita kehitettiin niin, että nollalla jakamisen tu-

lokseksi tuli 0. Samalla populaation koko kasvatettiin 500:n ja mutaatiot otettiin käyttöön (mutaatioprosentti 10%). Lisäksi toisessa testissä evaluointifunktioita ei kilpailutettu toisiaan vastaan vaan niiden suoriutuminen mitattiin vertaamalla niitä evaluointifunktioon *weight*10.4. Tällä menetelmällä saatiin aikaan kehitystä. Jotkin satunnaiset vertailut evaluointifunktioon *weight*10.4 tuottivat sukupolvien parhaiden yksilöiden voittoprosenteiksi yli 50%, vaikka kehitys kauttaaltaan oli vaatimattomampaa [Sii02].

Kolmannessa testissä evaluointifunktioiden vertailuissa pelattiin ennalta määrättyjä pelejä. Täten riitti, että vertailuevaluointifunktio *weight*10.4 pelasi 2 peliä jokaista sukupolven evaluointifunktiota vastaan. Tämän ansiosta populaation koko voitiin kasvattaa 2000:een. Mutaatioprosenttia nostettiin 30%:iin. Kolmannessa testissä saatiin aikaan kehitystä suhteessa vertailuevaluointifunktioon. Paras testissä kehitetty evaluointifunktio pääsi jopa pelikiekkoprosenttiin 75%. Vaikka kehitystä tapahtui suhteessa käytettyyn mittaan, ei yleisempää kehitystä juuri tapahtunut. Kun ennalta määrätyt pelit vaihdettiin satunnaisiin, kehityskäyrät muuttuivat vaakasuoriksi.

Tutkielman geneettisillä ohjelmoinneilla ei saatu tuotettua niin hyviä evaluointifunktioita kuin optimoinneilla saatiin. On mahdollista, että jokin tai jotkin geneettisen ohjelmoinnin piirteistä eivät olleet suotuisia kehittymiselle. Geneettisen ohjelmoinnin parametrien valinta saattoi hidastaa kehitystä. Kolme tehtyä testiä ei voi kattaa kuin murto-osan erilaisista parametrivalinnoista ja toteutus sekä laskenta-aika asettivat rajoitteita valintoihin. Toteutuksen tehokkuutta olisi voitu ehkä parantaa. Esimerkiksi risteytys olisi voitu toteuttaa jollain tehokkaammalla tavalla. Samoin evaluointifunktioita ja mutaatioita olisi voitu kehittää edelleen.

Moriarty ja Miikkulainen esittävät artikkelissaan tavan koodata neuroverkkoevaluointifunktion ikään kuin DNA:na [Mor93]. Tällöin risteytykset tapahtuvat tässä keinotekoisessa DNA:ssa ja ne saattavat yhdistää kahden yksilön hyviä ominaisuuksia paremmin kuin tutkielmassa esitelty raaka kahden evaluointifunktion osien vaihtaminen. Tutkielmassa toteutetut geneettisen ohjelmoinnin osat ovat kuitenkin sellaisia kuin lähdekirjallisuudessa [Ban98, Nil98] ja niillä on saatu hyviäkin tuloksia.

Tutkielman tuloksia voidaan verrata Gabriel Ferrerin vastaaviin tuloksiin. Hän on Master of Science -tutkielmassaan koittanut parantaa Othellon evaluointifunktiota käyttäen geneettistä ohjelmointia [Fer96]. Hänen evaluointifunktionsa ja geneettisen ohjelmoinnin toteutus ovat osittain erilaisia kuin tässä tutkielmassa.

Evaluointifunktiot kehittyivät Ferrerin tutkielmassa hieman paremmiksi kuin satunnaiset pelaajat, mutta hävisivät hyvin yksinkertaisia taktiikoita käyttäville tietokonepelaajille [Fer96, sivu 33]. Täten tutkielmassa savutettuja tuloksia voidaan pitää hyvinä suhteessa Ferrerin tuloksiin.

6.2 Saavutettu pelitaso

Tässä luvussa on tarkoitus selvittää minkälainen pelitaso tutkielmassa aikaansaaduilla evaluointifunktioilla saavutetaan. Tämä tehdään peluuttamalla parasta aikaansaatua evaluointifunktiota erilaisia vastustajia vastaan. Parhaaksi funktioksi on valittu *weight*10.4. Se on eräs parhaista optimointitestien tuloksena saaduista evaluointifunktioista keskinäisten vertailujen mukaan. Geneettisellä ohjelmoinnilla aikaansaadut evaluointifunktiot eivät pääse evaluointifunktion *weight*10.4 tasolle, kuten luvussa 5 nähtiin. Geneettisessä ohjelmoinnissa tapahtuva kehitys oli enemmänkin pientä sukupolvien laajuista kehitystä, kun taas optimointitesteissä saatiin aikaan huippuyksilöitä.

Evaluointifunktiota *weight*10.4 on peluutettu neljää erilaista vastustajaa vastaan. Kaikki vastustajat olivat WWW-sivuilla toimivia Othello-ohjelmia, jotka löydettiin linkkisivun [Haa02] kautta. Pelejä pelattiin siten, että tietokone laski yhden siirron eteenpäin eikä tunnistanut pelin lopputilanteita. Tällöin joka tilanteessa siirrettiin aina se siirto, jonka evaluointifunktio määräsi parhaaksi. Jos parhaita siirtoja oli useampia, arvottiin niiden kesken tehtävä siirto. Tällä pelitavalla ei saavutettu parasta mahdollista pelitasoa, mutta maksimoitiin evaluointifunktion vaikutus pelin kulkuun.

Pelejä ei pystytty automatisoimaan vaan ne jouduttiin pelaamaan siirtämällä kukin siirto käsin käyttöliittymän kautta. Koska pelaaminen tällä tavoin oli hidasta, tyydyttiin pelaamaan vain kuusi peliä kutakin vastustajaa vastaan. Näin pieni pelien määrä antaa vain suunnan sille mikä on todellisuudessa pelaajien keskinäinen paremmuusjärjestys. Kuusi peliä pelattiin vaihtaen joka pelin jälkeen aloittajaa mikäli vastustajaohjelman toteutus mahdollisti tämän. Taulukkoon 14 on koottu kunkin vastustajan osalta kuuden pelatun pelin lopputulokset ja voittosuhteet. Pelien tuloksissa ensimmäinen luku on evaluointifunktion *weight*10.4 pelikiekkojen määrä pelin lopputilanteessa ja toinen luku vastustajan pelikiekkojen määrä. Voittosarakkeessa ensimmäinen luku ilmoittaa evaluointifunktion *weight*10.4 voittojen määrän.

vastustaja	**1. peli**	**2. peli**	**3. peli**	**4. peli**	**5. peli**	**6. peli**	**voitot**
CS536	38 – 26	44 – 20	45 – 19	38 – 26	30 – 34	46 – 18	5 – 1
Diana	22 – 42	41 – 23	56 – 8	30 – 34	49 – 15	42 – 22	4 – 2
Nethello (1)	28 – 36	47 – 17	47 – 17	31 – 33	38 – 26	47 – 17	4 – 2
Nethello (2)	31 – 33	20 – 44	34 – 30	31 – 33	21 – 43	21 – 43	1 – 5
Nethello (3)	23 – 41	17 – 47	12 – 52	11 – 53	19 – 45	27 – 37	0 – 6
Ajax (Dummy)	33 – 31	21 – 42	38 – 26	30 – 34	22 – 42	44 – 20	3 – 3
Ajax (Beginner)	23 – 41	11 – 53	7 – 57	3 – 61	22 – 42	24 – 39	0 – 6

Taulukko 14: *Evaluointifunktion weight*10.4 *pelaamien pelien tuloksia eri vastustajia vastaan (muodossa weight*10.4 *– vastustaja).*

Ensimmäinen vastustaja [CS536] oli Othelloa pelaava oppiva ohjelma, joka on tehty vuonna 1996 Rutgersin yliopiston kurssilla *CS536: Machine Learning*. Ohjelma käyttää palauteoppimista (*reinforcement learning*) säätääkseen evaluointifunktionsa painoarvoja [You96, sivut 1-3]. Ohjelman evaluointifunktio laskee yhteen tietyillä painokertoimilla painotettuna kulma-, reuna- ja keskusruutujen määrän sekä omien ja vastustajan mahdollisten siirtojen määrän [You96]. CS536 pelaa suoraan evaluointifunktionsa ilmoittaman siirron. Tämän ansiosta pelatut pelit vertasivat käytännössä evaluointifunktioiden paremmuutta. Myös itse evaluointifunktiot olivat hyvin vertailtavia, koska molemmat on tuotettu jollain oppimismenetelmällä. CS536 ei mahdollistanut aloittajan vaihtoa, joten kaikki kuusi peliä pelattiin evaluointifunktio *weight*10.4 aloittaen. Vastustaja hävisi kaikki muut pelit paitsi yhden. Tulosten perusteella voidaan päätellä, että evaluointifunktio *weight*10.4 on parempi kuin CS536:n evaluointifunktio.

Toinen vastustaja oli Java-ohjelma nimeltä Diana [DIANA]. Kuten CS536, sekään ei laske siirtoja eteenpäin vaan tekee siirrot suoraan evaluointifunktionsa mukaan. Dianan evaluointifunktio on saman tyyppinen kuin *weight*10.4. Siinäkin eri ruuduille annetaan eri painoarvoja ja niitä lasketaan yhteen. Dianan evaluointifunktiossa painoarvot on kuitenkin säädetty käsin. Diana mahdollistaa molemmilla väreillä pelaamisen, joten kumpikin pelaaja pelasi kolme peliä mustilla ja kolme valkoisilla. Peleistä *weight*10.4 voitti 4 ja Diana 2. Pelikiekkojen yhteismäärässä Diana oli selkeästi huonompi. Tämän perusteella voidaan olettaa, että *weight*10.4 on Dianan evaluointifunktiota hieman parempi.

Kolmas vastustaja oli Internetissä pelattava Nethello [NETHELLO]. Sen peli perus-

tuu siirtojen eteenpäin laskemiseen. Nethello pystyy laskemaan yhdestä kymmeneen siirtoa eteenpäin. Evaluointifunktiota *weight*10.4 peluutettiin Nethellon kolmea alinta tasoa vastaan (taulukon 14 rivit 3,4 ja 5). Näillä tasoilla Nethello laskee yksi, kaksi ja kolme siirtoa eteenpäin. Nethellossa ei ole mahdollista vaihtaa aloittajaa vaan ainoastaan värejä. Tästä syystä kaikki pelit pelattiin niin, että evaluointifunktio *weight*10.4 aloitti. Yhden siirron eteenpäin laskeva Nethello voitti kaksi peliä kuudesta, minkä perusteella Nethellon käyttämä evaluointifunktio näyttäisi olevan hieman huonompi kuin *weight*10.4. Kun Nethello laski 2 siirtoa eteenpäin, voitti *weight*10.4 enää vain yhden pelin kuudesta. Nethellon laskiessa kolme siirtoa eteenpäin, voitti se kaikki pelit ja vieläpä melko selkeästi. Tästä voidaan havaita, että evaluointifunktion ja eteenpäin laskemisen yhdistäminen tulee nopeasti pelkkää evaluointifunktiota kannattavammaksi.

Neljäs ja viimeinen vastustaja oli Java-sovelma Ajax [AJAX], joka on luultavasti parhaiten pelaava on-line Othello-ohjelma tällä hetkellä (7/2003). Ajax on Othello-ohjelman Edax kevyempi, Internetissä pyörivä versio. Edax on voittanut vuonna 2000 tietokoneiden Ranskan mestaruuden Othellossa. Ajaxissa on viisi pelitasoa, joista kaksi helpointa ovat Dummy ja Beginner. Vertailupelit pelattiin näitä tasoja vastaan (taulukon 14 kaksi alimmaista riviä). Ajax mahdollistaa aloittajan vaihdon, joten joka toisen pelin aloitti *weight*10.4 ja joka toisen Ajax. Tasolla Dummy Ajax voitti puolet peleistä, joten evaluointifunktiolla *weight*10.4 saavutettu pelitaso on kutakuinkin samaa tasoa kuin Ajaxin alin taso. Beginner-tasolla Ajax voitti kaikki pelit selkeästi. Pelien alussa tilanne näytti tasaväkiseltä ja evaluointifunktiolla *weight*10.4 oli enemmän kiekkoja kuin Ajaxilla. Lopussa kaikki pelit kääntyivät kuitenkin selkeäksi Ajaxin voitoksi. Tämä johtunee siitä, että Ajax on erityisen vahva loppupelissä ja pystyy estämään vastustajan siirrot.

Edellä mainitut tulokset antavat hieman viitettä siitä kuinka hyvä evaluointifunktio *weight*10.4 on. Vertailussa Internetistä löydettyihin, vain evaluointifunktiota käyttäviin Othello-ohjelmiin *weight*10.4 pärjää hyvin. Näiden evaluointifunktiot ovat usein hyvin yksinkertaisia ja intuitiivisia funktioita, joissa lasketaan kertoimilla painotettuna eri ruuduissa olevien pelikiekkojen ja mahdollisten siirtojen määrää. Jos vastustajaohjelma laskee siirtoja eteenpäin, ei *weight*10.4 enää pärjääkään niin hyvin.

On luonnollista, että pelkästään evaluointifunktioon perustuvassa pelissä voidaan saavuttaa vain tietty taso. Ainoastaan pelilaudan tilanteen katsominen ja siirtäminen tutkimatta tulevia siirtoja ei ole Othellossa kovin järkevä tapa pela-

ta. Teoriassa evaluointifunktio voi "nähdä" oikean siirron pelilaudan tilanteesta, mutta käytännössä tällaisen evaluointifunktion toteuttaminen ei liene mahdollista. Evaluointifunktio *weight*10.4 ei ole niin hyvä, että se voittaisi Othello-ohjelmat, jotka laskevat siirtoja eteenpäin. Jo kahden siirron eteenpäin laskeminen auttoi Nethelloa voittamaan sen.

Internetistä löytyvien on-line Othello-ohjelmien taso ei ole kovin korkea. Joitain poikkeuksia, kuten Ajax ja Nethello, lukuun ottamatta aloittelijakin voittaa ne melko helposti. Tutkielmassa toteutettua ohjelmaa pääsee testaamaan tutkielman sovelluksen WWW-sivulla [IMPOTHELLO]. Siellä on Java-sovelma, joka tekee siirtonsa evaluointifunktion *weight*10.4 mukaan. Valittavana on kuinka monta siirtoa (1 – 4) eteenpäin sovellus laskee eli mikä on minmax -algoritmin maksimisyvyys. Muutamien testipelien pohjalta näyttäisi siltä, että sovelman laskiessa 4 siirtoa eteen päin, pystyy se pelaamaan tasaväkisesti aloittelijan kanssa.

6.3 Pelit ja koneoppiminen

Pelit tarjoavat oivan alustan koneoppimismenetelmille. Tässä luvussa kuvataan kuinka koneoppimismenetelmiä voidaan hyödyntää peliohjelmistojen kehittämisessä. Aluksi esitellään kaksi tunnettua peliohjelmistoa ja niiden toteutusperiaatteita. Sen jälkeen ohjelmistojen kehittämistä ja koneoppimisen mahdollisuuksia siinä tarkastellaan yleisemmin.

Usein parhaat peliohjelmat perustuvat oppimisen sijasta raakaan laskentaan. Esimerkiksi Gari Gasparovin (shakin maailman mestari) voittaneen tietokoneen, Deep Bluen, menestys perustuu pääosin laitteistoon, joka kykenee laskemaan siirtoja eteenpäin nopeammin kuin mikään ohjelmisto [Cam02, sivu 58]. Deep Blue voitti vuonna 1996 Gasparovin $3,5 - 2,5$ [Sch97]. Laitteiston lisäksi Deep Bluessa on laajat tietokannat, jotka sisältävät suuren määrän pelin teoriaa, aloituksia, pelattuja pelejä ja loppupelejä [Cam02]. Kaikki massiivinen laskenta on saatu niin tehokkaaksi, että tietokone pystyy pelaamaan shakkia reaaliaikaisesti.

Maailman tällä hetkellä (7/2003) parhaassa Othello-ohjelmistossa, Logistellossa on hieman erilainen periaate kuin Deep Blue:ssa [Bur97b, sivut 1-3]. Oleellisin ero on, että Logistello hyödyntää oppimista. Logistello käyttää aloituskirjastoa (*opening book*), jonka se on muodostanut useista itsensä ja muiden pelaamista peleistä. Logistello päivittää koko ajan kirjastoaan, joten se ei häviä peliä koskaan samalla tavalla ja oppii paremmaksi pelatessaan [Bur97b, sivut 1-3]. Evaluointifunktio-

na Logistellossa käytetään erilaisia ominaisuuksia yhdistelevää lineaarista funktiota. Lisäksi se hyödyntää pelipuun läpikäynnissä erilaisia tehostusmenetelmiä. Vuonna 1997 Logistello voitti hallitsevan maailmanmestarin Takeshi Murakamin 6 – 0 [Bur97a].

Deep Blue:ta ja Logistelloa verrattaessa, Logistello on menestynyt hieman paremmin. Syynä on, että Othello on shakkia yksinkertaisempi peli ja siinä pärjää helpommin raa'alla laskemisella. Onkin arvioitu, että vuonna 2010 Othello on ratkaistu [Her02, sivu 306]. Tämä tarkoittaa, että pelin lopputulos tiedetään jo pelin alkaessa, kun molemmat pelaajat pelaavat parhaalla mahdollisella tavalla.

Yleisesti ottaen koneoppimismenetelmät ovat tehottomampi keino kehittää peliohjelmistoja kuin suora ohjelmiston toteutus. Useat huippuohjelmistot kuitenkin käyttävät erilaisia oppimismenetelmiä esimerkiksi evaluointifunktioiden kehittämiseen [Für01, sivut 9-24] tai pelitietokannan ylläpitämiseen [Für01, sivut 5-6]. Oppimismenetelmiä käytetään sekä ohjelmistojen kehittämisessä että niiden osana.

Kaikkein käytetyin tapa peliohjelmistojen parantamiseksi on evaluointifunktion parantaminen [Für01, sivu 9]. Sen voi tehdä hyvin monella eri tavalla. Pelejä voi pelata tiettyjä vastustajia vastaan, keskittyä joihinkin merkittäviin pelitilanteisiin tai vaikkapa pyrkiä saamaan tietystä pelitilanteesta jokin oikea evaluoinnin arvo [Für01, sivut 9-24]. Evaluointifunktion parantaminen ei kuitenkaan ole ainoa tapa ohjelmistojen parantamiseksi. Ohjelmisto voi oppia pelaamaan paremmin esimerkiksi oppimalla tunnistamaan pelilaudan tai pelin tiettyjä kuvioita, muistamalla tekemiään virheitä ja välttelemällä niitä tai oppimalla uusia heuristiikkoja pelipuun läpikäynnissä [Für01].

6.4 Pohdintaa

Tutkielmassa käsiteltyjä evaluointifunktion parantamismenetelmiä ei tällaisinaan luultavasti ole koitettu aiemmin. Geneettisellä ohjelmoinnilla tosin on yritetty parantaa Othellon evaluointifunktiota [Fer96], mutta tämä on toteutettu hieman eri tavalla kuin tutkielmassa. Optimointiongelmaksi evaluointifunktion parantamisongelmaa ei luultavasti olla käännetty. Optimointiongelman ratkaisu voidaan kuitenkin tulkita jonkinlaiseksi palauteoppimiseksi, jossa evaluointifunktion vakioiden arvoja muutetaan sen pelimenestyksestä saaman palautteen mukaan. Erilaisia palauteoppimista käyttäviä menetelmiä on kokeiltu evaluointi-

funktioiden parantamismenetelminä [Für01, sivut 14-16], joten optimoinninkin kaltaisia menetelmiä on testattu.

Optimoinnin käyttäminen jossain muodossa voisi olla järkevää tarkasti suunnitellun evaluointifunktion painoarvojen säätämisessä. Sen kehittämisessä on luultavasti potentiaalia parempiinkin tuloksiin kuin mitä tutkielmassa saatiin. Tieto kyseessä olevasta pelistä voisi parantaa optimoinnin tehokkuutta merkittävästi.

Geneettisissä ohjelmoinneissa populaatioiden koot ovat usein suurempia kuin tässä tutkielmassa suoritetuissa testeissä. Yhteen evaluointifunktioiden vertailuun kuluva aika oli tutkielman toteutuksessa niin suuri, että populaation koko juoduttiin pitämään pienenä. Geneettinen ohjelmointi saattaisi tuottaa parempia tuloksia, jos vertailu saataisiin riittävän nopeaksi ja tätä kautta populaation koko kasvamaan.

Tässä tutkielmassa esitellyt oppimismenetelmät paransivat evaluointifunktioita, mutta vain tiettyyn rajaan asti. Ne eivät vaikuttaneet tehokkuudeltaan kovin hyviltä. Ne veivät suhteessa aikaa melko paljon, mutta kehittivät evaluointifunktiota kuitenkin vain vähän. Tämä pätee erityisesti geneettiseen ohjelmointiin, joka vei huomattavasti enemmän laskenta-aikaa kuin optimointi, mutta tuotti kuitenkin huonompia evaluointifunktioita.

Lähteet

Bai98 Baird D., Gertner R., Picker R., *Game theory and the law*, Harvard University Press, Cambridge Mass., 1998.

Ban98 Banzhaf W., *Genetic programming - an introduction : on the automatic evolution of computer programs and its applications*, Morgan Kaufmann, San Francisco, CA, 1998.

Bur63 Burger E., *Introduction to the Theory of Games*, Prentice-Hall, Englewood Cliffs, NJ, 1963.

Bur97a Buro M., *The Othello Match of the Year: Takeshi Murakami vs. Logistello*, ICCA Journal, volume 20, numero 3, sivut 189-193, 1997.

Bur97b Buro M., *An Overview of Logistello*, tekijän lyhyt selitys Logistellon toteutusperiaatteista, 1997. Löytyy Michael Buron julkaisusivulta *http://www.cs.ualberta.ca/~mburo/publications.html* (7/2003).

Bur97c Buro M., *Experiments with Multi-ProbCut and a new high-quality evaluation function for Othello*, Workshop on game-tree search, NECI, 1997.

Cam02 Campbell M., Hoane J., Hsu F., *Deep Blue*, Artificial Intelligence, volume 134, sivut 57-83, 2002.

Fer96 Ferrer G., *Using genetic programming to evolve board evaluation functions*, Master's thesis, Department of Computer Science, School of Engineering and Applied Science, University of Virginia, Charlottesville, VA, 1996.

Fle81 Fletcher R., *Practical methods of optimization*, volume 2, Wiley, Chichester, 1981.

Für01 Fürnkranz J., *Machine learning in games: A survey*, kirjasta *Machines that Learn to Play Games*, luku 2, sivut 11-59, Nova Science Publishers, Huntington, NY, 2001.

Gil81 Gill P., Murray W., Wright M., *Practical optimization*, Academic Press, London, 1981.

Haa02 Haan F., Linkki-sivu Java/HTML Othello-ohjelmiin, WWW-osoite *http://www.frankdh.demon.nl/othello/java_html_othello.html* (7/2003).

Her02 van den Herik J., Uiterwijk J., van Rijswijck J., *Games solved: Now and in the future*, Artificial Intelligence, volume 134, sivut 277-311, 2002.

Hyv93 Hyvönen E., Karanta I., Syrjänen M., Carlson L., *Tekoälyn ensyklopedia*, Gaudeamus, Helsinki, 1993.

Kiy87 Kiyosi I., *Encyclopedic dictionary of mathematics*, MIT Press, Cambridge, MA, 1987.

Kre98 Kreps D., *Game theory and economic modelling*, Clarendon press, Oxford, 1998.

Laa87 Laarhoven P., Aarts E., *Simulated annealing : theory and applications*, Reidel, Dordrecht, 1987.

Lue84 Luenberger D., *Linear and nonlinear programming*, Addison-Wesley, Reading, MA, 1984.

Mic96 Michalewicz Z., *Genetic algorithms + data structures = evolution programs*, Springer, Berlin, 1996.

Mor93 Moriarty D., Miikkulainen R., *Evolving complex Othello strategies using marker-based genetic encoding of neural networks*, Technical Report AI93-206, Department of Computer Sciences, University of Texas, Austin, USA, 1993.

Nil98 Nilsson N., *Artificial intelligence : a new synthesis*, Morgan Kaufmann, San Francisco, CA, 1998.

Oja00 Oja H., *Maailmankaikkeus 2001*, Tähtitieteellinen yhdistys Ursa, Helsinki, 2000.

Rud76 Rudin W., *Principles of mathematical analysis*, McGraw-Hill, New York, NY, 1976.

Rus95 Russell S., Norvig P., *Artificial intelligence : a modern approach*, Prentice-Hall, Englewood Cliffs, NJ, 1995.

Sch97 Schaeffer J., Plaat A., *Kasparov versus Deep Blue: The Re-match*, ICCA Journal, volume 20, numero 2, sivut 95-102, 1997.

Sii02 Siivonen S., Pro gradu -tutkielman tulosten WWW-sivu, osoite *http://www.cs.helsinki.fi/u/ssiivone/gradu* (7/2003), löytyy liitteestä 4.

Wik02 Wikla A., Tietorakenteet-kurssin oppimateriaali, syksy 2002, Helsingin yliopisto, Tietojenkäsittelytieteen laitos, osoite *https://www.cs.helsinki.fi/i/wikla/Tira/Sisalto/* (7/2003).

You96 Young P., *Reinforcement Experiments with Othello*, Seminaariesitelmä kurssilla CS536 : Machine Learning, Rutgers University, New Jersey, USA, 1996. Löytyy osoitteesta *http://paul.rutgers.edu/~pmyoung/MachineLearning/536paper.ps* (7/2003).

AJAX Ajax (versio 1.3), Othelloa ymmärtävä Java-sovelma, tehnyt Richard Delorme, Ranska, 2001. Löytyy osoitteesta *http://perso.club-internet.fr/abulmo/ajax-en.htm* (7/2003).

CS536 Oppiva on-line Othello-ohjelma, tehty kurssilla CS536: Machine Learning, Rutgers University, New Jersey, tehnyt Philip Young, USA, 1996. Löytyy osoitteesta *http://paul.rutgers.edu/~pmyoung/MachineLearning/* (7/2003).

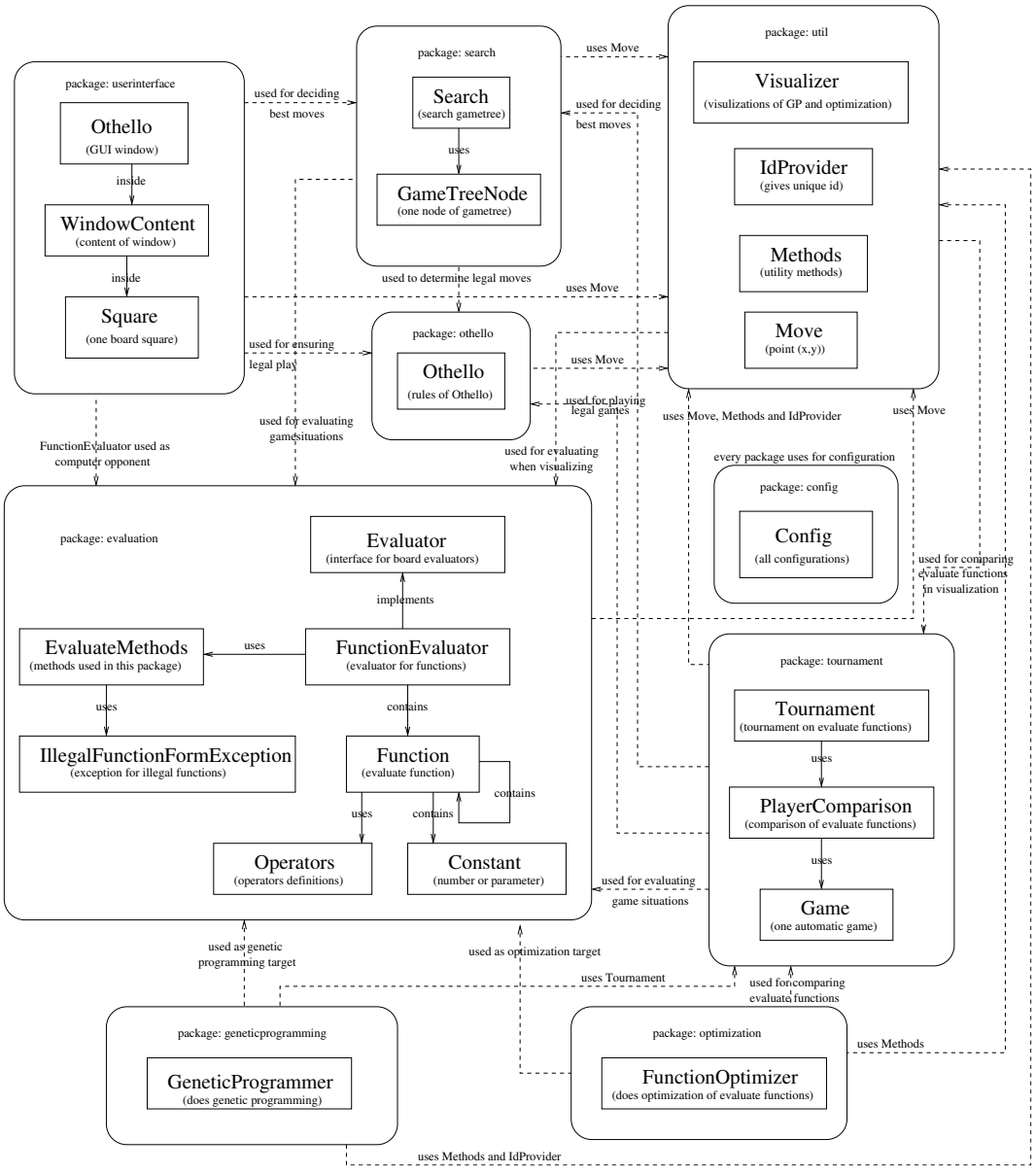
DIANA Diana, on-line Othello-ohjelma (Java-sovelma), tehnyt Johnny Norre, Tanska, 1998. Löytyy osoitteesta *http://users.cybercity.dk/~ccc9169/DuNoDiana.html* (7/2003).

IMPOTHELLO ImpOthello - Othelloa pelaava Java-ohjelma, tätä tutkielmaa varten tehty sovellus, tehnyt Samuli Siivonen, 2002. WWW-osoite *http://www.cs.helsinki.fi/u/ssiivone/gradu/impothello* (7/2003), löytyy myös liitteestä 4.

NETHELLO Nethello, kaikissa selaimissa pelattava CGI-Othello, tehnyt Niklas Eén ja Magnus Ihse, Ruotsi, 1997. Löytyy osoitteesta *http://janus.cs.chalmers.se/~www/nethello/* (9/2002).

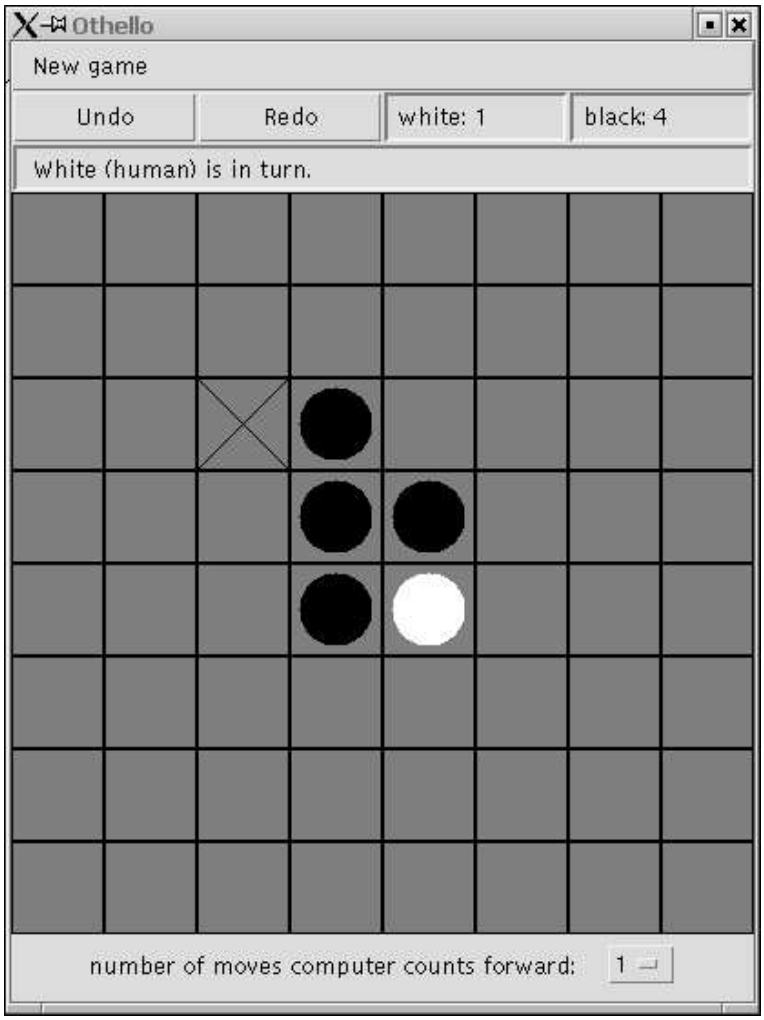
Liite 1. Sovelluksen arkkitehtuurikuvaus

Alla on tutkielman ohessa toteutetun sovelluksen, ImpOthellon, arkkitehtuurikuvaus. Kaikki luokat on kuvattu suorakaiteina ja niiden nimien jälkeen on lyhyt englanninkielinen kuvaus luokasta. Luokkien väliset yhteydet pakettien sisällä on kuvattu nuolilla ja niihin liittyvillä lyhyillä kuvauksilla. Paketit on kuvattu suorakaiteilla, joiden kulmat on pyöristetty. Pakettien väliset yhteydet on kuvattu katkonaisilla nuolilla ja näihin liittyvillä lyhyillä kuvauksilla. Luokkien ja pakettien tarkemmat kuvaukset löytyvät tutkielman sovelluksen WWW-sivulta [IMPOTHELLO].



Liite 2. Sovelluksen käyttöohjeet

Kuvassa 1 on tutkielman sovelluksen, ImpOthellon, käyttöliittymä omassa ikkunassaan. Käyttöliittymän saa auki ajamalla sovelluksen käyttöliittymäluokka (esimerkiksi komennolla `java userinterface.Othello`). Sovelluksella on myös toinen, WWW-sivulla toimiva käyttöliittymä [IMPOTHELLO], joka näyttää hyvin samanlaiselta. Seuraavat ohjeet koskevat omaan ikkunaansa aukeavaa ImpOthellon käyttöliittymää, mutta ne ovat suurelta osin sovellettavissa myös WWW-sivulla toimivaan käyttöliittymään.



Kuva 1: Kuva ImpOthellon käyttöliittymästä

Ylhäällä olevasta *New game* -valikosta voi valita haluamansa pelityylin, ladata talletetun pelin (*Load game*) tai poistua ohjelmasta (*Exit*). Pelityylejä on neljä erilaista:

ihminen ihmistä vastaan (*Human vs human*), tietokone ihmistä vastaan (*Computer starts*), ihminen tietokonetta vastaan (*Human starts*) ja tietokone tietokonetta vastaan (*Computer vs computer*). Pelin aloittaminen tapahtuu valitsemalla jokin pelityyleistä. Mustilla pelaava pelaaja aloittaa aina pelin. Siirtojen tekeminen tapahtuu ihmisten osalta hiiren avulla viemällä hiiren osoitin siirtoa vastaavan ruudun päälle ja painamalla hiiren nappia. Kun hiiren osoitin on jonkin laillisen siirtoruudun päällä, piirtyy ruudulle rasti kuten kuvassa 1. Tietokone siirtää itsestään, kun sen vuoro on.

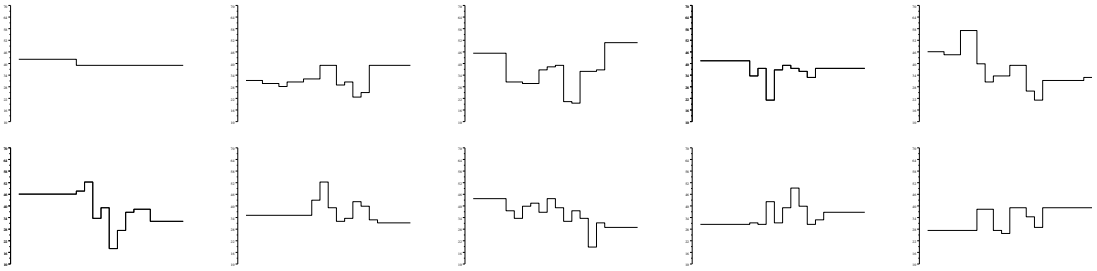
Ikkunan ylälaidassa on kaksi nappulaa (*Undo* ja *Redo*) sekä kaksi tekstikenttää. Tekstikentät ilmoittavat mikä määrä mustia ja valkoisia nappuloita on laudalla kussakin pelin tilanteessa. *Undo*-nappulalla voi peruuttaa tekemiään siirtoja aina pelin alkutilanteeseen saakka ja *Redo*-nappulalla voi tehdä uudelleen peruutetun siirron. Tällä tavalla pelin kulkua voi seurata siirto siirrolta nykyisestä siirrosta aina pelin alkuun asti. Jos pelaaja tekee siirron omalla vuorollaan selatessaan siirtohistoriaa, jatkuu peli tästä eteenpäin aivan normaalisti ja historiatiedot kyseisestä siirrosta eteenpäin korvautuvat uusilla.

Pelilaudan ja yläpalkin välissä on tekstikenttä, johon tulee tietoa pelitilanteesta, kuten kumman siirtovuoro on tai onko toinen voittanut pelin tai kykenemätön siirtämään. Pelilaudan alapuolella on teksti *number of moves computer counts forward* ja valintapalkki, jossa on valittavana numerot 1 – 4. Valittu numero ilmoittaa kuinka monta siirtoa tietokone laskee eteenpäin, kun se päättää omaa siirtoaan. Valittua numeroa voi vaihtaa kesken pelin. Jokaisen tietokoneen siirron aikana se laskee niin monta siirtoa eteenpäin, kuin valintapalkin numero ilmoittaa.

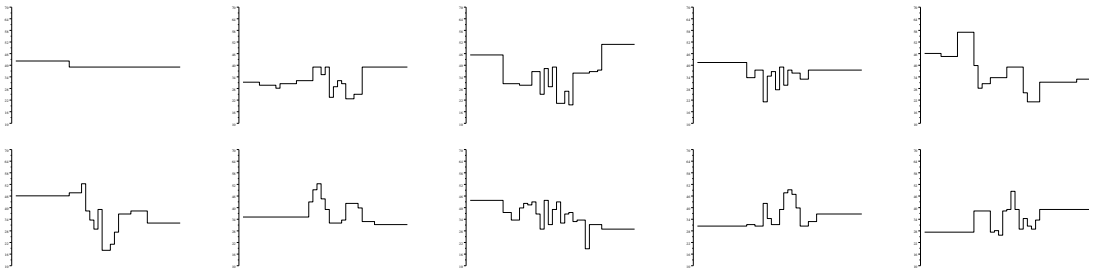
New game -valikosta valittava pelin lataaminen (*Load game*) tarkoittaa tiedostossa `loadGame` olevien siirtojen lukemista. Tässä tiedostossa on allekkain kaikki pelissä tehdyt siirrot. Luokka `tournament.Game` luo tämän muotoisia tiedostoja, kun se tallettaa automaattisesti pelattuja pelejä. Kun käyttöliittymään ladataan peli, se pelaa tiedostossa olevat siirrot kuin kaksi ihmispelaajaa pelaisi vastaavan pelin. Tämän jälkeen peli on käyttöliittymän siirtohistoriassa ja sen etenemistä voi tutkia *Undo* ja *Redo* nappuloiden avulla.

Liite 3. Optimoinnin kohdefunktion kuvaajia

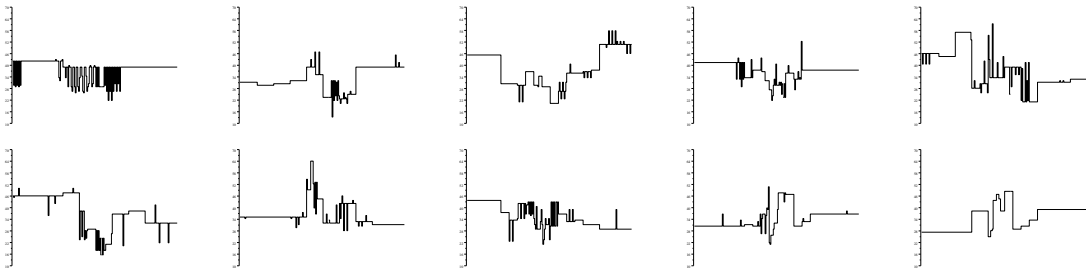
Luvussa 4.3.4 on esitelty osa kuvaajista, jotka kuvaavat ei-satunnaisten optimointien kohdefunktiota. Tässä liitteessä esitellään kuvaajat kaikkien painoarvojen osalta. Kuvaajat on laskettu muuttamalla kutakin evaluointifunktion *weight*10 painoarvoa tietyn askelmäärän verran ja laskemalla sen pelikiekkoprosentti vertailussa evaluointifunktioon *weight*10.4. Alla olevissa kuvissa ovat kutakin evaluointifunktion painoarvoa vastaavat kuvaajat järjestyksessä vasemmalta oikealle ensin ylärivi ja sitten alarivi.



Kuva 1: Painoarvoja vastaavat kuvaajat askelvälillä 1.



Kuva 2: Painoarvoja vastaavat kuvaajat askelvälillä 0, 5.



Kuva 3: Painoarvoja vastaavat kuvaajat askelvälillä 0, 1.