Parviälytekniikoita
– Staattisessa evaluointifunktiossa käytettävän neuroverkon optimointi PSO:lla

Henrik Huttunen
Pro gradu -tutkielma
Tietojenkäsittelytiede
Informaatioteknologian laitos
TURUN YLIOPISTO
2008

TURUN YLIOPISTO
Informaatioteknologian laitos

HUTTUNEN, HENRIK: Parviälytekniikoita
– Staattisessa evaluointifunktiossa käytettävän neuroverkon optimointi PSO:lla

Pro gradu -tutkielma, 78 s., 5 liites.
Tietojenkäsittelytiede
Toukokuu 2008

Parviäly on tekoälymuoto, jossa joukko yksinkertaisia agentteja ratkaisee algoritmisen ongelman ilman, että yksittäinen agentti ymmärtää toimintansa vaikutuksen kokonaisuuden kannalta. Ratkaisu muodostuu *itseorganisoidusti*, eli ilman ulkoista ohjausta. Ominaista on myös, että yksilöt mahdollisesti viestivät epäsuorasti muokkaamalla ympäristöä (*stigmergy*).

Tutkielmassa käsitellään pääosin optimointiongelmia ratkaisevia metaheuristiikkoja, jotka ovat muurahaisoptimointi (ACO), stokastinen diffuusiohaku (SDS) ja parvioptimointi (PSO). Erityisesti tarkastellaan erilaisia parvioptimoinnin tekniikoita, jotka ovat perusversiota kehittyneempiä tai ovat erikoistuneet tiettyyn sovellusalueeseen, kuten dynaamiseen ympäristöön. Myös lukuisia usean optimin ongelmien ratkaisuun erikoistuneita menetelmiä käsitellään niiden tärkeyden vuoksi.

Tässä työssä tutkitaan, kuinka tekoälyn kehittäminen onnistuu Connect-lautapelille, kun käytetään rinnakkaisevoluutiota ja kolmea eri PSO-tekniikkaa – perusversiota, FIPSiä ja CLPSO:ta. PSO:lla optimoidaan tekoälyn staattisessa evaluointifunktiossa käytettävän neuroverkon painoja – painojen suhteellinen hyvyys saadaan selvitettyä pelauttamalla tekoälyjä vastakkain.

Tekoälypelaajia testattiin Connect-pelille etupäässä 3x3-kokoisella pelilaudalla. Työssä tarkastellaan eri parametrien, kuten PSO-kehitysiteraatiomäärän ja parven koon vaikutusta kehitetyn pelaajan hyvyyteen. Referenssipelaajina käytettiin satunnaisen siirron tekevää, naiivia logiikkaa sisältävää ja minimax-algoritmia käyttävää pelaajaa.

Tulokset ovat tilastollisesti pääosin vain suuntaa antavia. On kuitenkin selvää, että esimerkiksi iteraatiomäärän kasvattaminen auttaa paremman tekoälypelaajan kehittämisessä. Muodostetut pelaajat ovat myös usein kilpailukykyisiä referenssipelaajia vastaan. Lisäksi testatuista tekniikoista lähes johdonmukaisesti parhaita tuloksia antava PSO on FIPS (Fully Informed Particle Swarm).

Avainsanat:

parviäly, metaheuristiikka, PSO, ACO, SDS, rinnakkaisevoluutio, Connect-4

UNIVERSITY OF TURKU
Department of Information Technology

HUTTUNEN, HENRIK: Swarm Intelligence Techniques
– Optimization of the Neural Network of a Static Evaluation Function with PSO

Master's thesis, 78 p., 5 appendix p.
Computer Science
May 2008

Swarm Intelligence is a form of Artificial Intelligence that has taken its inspiration from nature. The idea is to have a group of simple agents that try to solve a problem without knowing the global effects of their actions. The solution emerges by self-organization without the help of an external control. Another important aspect is that the agents possibly communicate through the modification of the environment, which is called stigmergy.

In this thesis, three optimization metaheuristics are discussed. They are Ant Colony Optimization (ACO), Stochastic Diffusion Search (SDS), and Particle Swarm Optimization (PSO). PSO is discussed in full length; different techniques that are usually better than the basic version are considered. Some of the techniques are focused on a specific domain, such as dynamic environment. Moreover, a plenty of PSO methods that deal with multiple optima are covered in detail for their importance.

Three PSO methods are tested to see whether it is possible to develop a decent AI player for the Connect board game using co-evolution. In the tests, the static evaluation function of a player is virtually just a neural network. To develop a better player, the weights of the neural networks are optimized using PSO, and the relative fitness of the players are decided by having them to play against each other.

Most of the tests are performed on a 3x3 sized board. It was studied how different parameters, e.g. the amount of iterations and the size of the swarm, affect the outcome. The three reference players for determining the quality of the artficial PSO players were a randomly playing AI, a player with naive logic, and a player using a minimax algorithm.

The results are not statistically reliable enough to make major conclusions, but it is clear that the amount of iteration has a considerable positive effect on the AI's performance. In addition, the developed players tend to perform well against the reference players. Moreover, compared with the basic version and CLPSO (Comprehensive Learning Particle Swarm Optimizer), FIPS (Fully Informed Particle Swarm) almost consistently yields better results.

Keywords:

swarm intelligence, metaheuristic, PSO, ACO, SDS, co-evolution, Connect-4

Sisällysluettelo

1 Johdanto **1**

2 Parviälyn perusteita **4**

2.1 Yleistä parviälystä *4*

2.2 Parviälymenetelmiä *5*

3 Parviälymetaheuristiikkoja **8**

3.1 Optimointiongelma *8*

3.2 Heuristiikat *10*

3.3 Muurahaisoptimointi *14*

3.4 Stokastinen diffuusiohaku *19*

4 Parvioptimointi (PSO) **24**

4.1 Perusidea *24*

4.2 Parametrien arvot perusversiossa *26*

4.3 Naapurustotopologiat *28*

4.4 Yliammuntaongelma *30*

4.5 Monitavoiteongelmat *32*

4.6 Dynaamiset ongelmat *32*

4.7 Adaptiivisuus *33*

4.8 Rajoiteoptimointi *37*

4.9 PSO:n analysointi *37*

4.10 Käytännön sovelluksia *38*

5 Parvioptimointi ja usean optimin ongelmat **40**

5.1 Yleistä *40*

5.2 Lokerointimenetelmät (NichePSO, SPSO) *40*

5.3 Gregarious-PSO (G-PSO) *41*

5.4 FER-PSO ja FDR-PSO *42*

5.5 SEPSO, CRS ja CRIBS *43*

5.6 Attractive-Repulsive PSO (ARPSO) 46

5.7 Breeding Swarms 47

5.8 Predator-Prey Optimization 48

5.9 Comprehensive Learning PSO (CLPSO) 49

5.10 HPSO-TVAC 49

5.11 Gaussian-Dynamic Particle Swarm (GDPS) 50

5.12 Monisuppiloiset ongelmat 51

6 Ongelmanratkaisu parviälyllä 53

6.1 Tehtävänasettelu 53

6.2 Neuroverkko 53

6.3 Connect-4-lautapeli 55

6.4 Neuroverkon opettaminen rinnakkaisevoluutiolla käyttäen PSO:ta 57

6.5 Testimenetelmä 58

6.6 Tulokset 63

6.7 Tulosten analysointi 68

6.8 Yhteenveto 71

7 Pohdinta 72

Viitteet 74

Liite 1. Kuvia saaduista tuloksista 79

1 Johdanto

Tekoäly on merkittävä tutkimusaihe jo pelkästään filosofiselta kannalta. Peruskysymyksenä on, voidaanko kehittää tietokone, jonka rajapinnan kautta havannoitu käyttäytyminen on erottamaton ihmisen käyttäytymisestä (Turingin testi). Toisaalta hyödyllisiä käytännön sovelluksia on rajattomasti. Niistä monet ovat optimointiongelmia; pyritään löytämään annettuun ongelmaan paras mahdollinen ratkaisu lukuisista esiinnousevista vaihtohdoista. Eräs tekoälyn paljon tutkittu sovellusalue on pelit, joissa tekoälyä käyttävä tietokoneohjelma pyrkii valitsemaan optimaalisen siirron vuorollaan. Tietokonepelaaja usein kehitetään tiettyä peliä varten, ja niinpä tällaisen ohjelman uudelleenkäyttö on vaikeaa. Esimerkiksi shakkipelin tekoälylle voidaan sisällyttää etukäteistietoa pelistä määräämällä nappuloiden arvioidut absoluuttiset arvot. [Nor03] Onkin luontevaa kysyä, onko tällainen tekoäly varsinaisesti älykäs. Kenties todellinen älykkyys on sen sijaan sellaista, että pelaaja kehittyy taitavaksi omin keinoin pelaamalla ja muuttamalla käyttäytymistään menestymisensä perusteella – tietämättä pelistä etukäteen mitään.

Tässä tutkielmassa sovelletaan aiemmin kehitettyä [Mes04] tekoälyn kehittämismenetelmää, jossa tekoälypelaaja ei ennen ohjelman suoritusta tiedä Connect-lautapelistä muuta kuin tavan saada pelitilanteen sallitut siirrot. Hyvää tekoälyä etsitään kilpailuttamalla neuroverkkoja käyttäviä tekoälypelaajia vastakkain. Tekoälypelaajan toiminnan määrää neuroverkon antama arvio laillisten siirtojen jälkeisille pelilaudan tilanteille; pelaaja valitsee siirron, joka päätyy parhaaseen siirtoon. Tekoälyä soveltavat ohjelmat pelaavat vastakkain useita kertoja ja keräävät pisteitä itselleen voitoilla ja tasapeleillä, joiden perusteella valitaan eniten pisteitä saanut pelaaja. Parasta tekoälyä etsitään hakemalla neuroverkon painoille arvoja parvioptimoinnilla, kunnes kehityskertoja on suoritettu ennalta annettu määrä. [Mes04, Mes06]

Tämän tutkielma testeissä käytetty parvioptimointi on eräs parviälyn osa-alue. Parviälyssä yhden älykkään agentin sijaan ratkaisua etsii useita agentteja, jotka yleisesti eivät ymmärrä toimintansa vaikutuksia kuin paikallisesti. Erityistä on, että lopullinen ratkaisu muodostuu yksilöiden yhteisvaikutuksesta. Tällä on suuri etu mm. skaalautuvuuden ja uudelleenkäytettävyyden vuoksi. [Ken01]

Tässä työssä käsitellään kolmea erilaista parviälymetaheuristiikkaa – muurahaisoptimointi (ACO, Ant Colony Optimization) [Dor04], stokastinen diffuusiohaku (SDS, Stochastic Diffusion Search) [DeM04] ja parvioptimointi (PSO, Particle Swarm Optimizer) [Ken01] – sekä esitellään lyhyesti muutamia muita parviälytekniikoita. Ensimmäisessä luvussa kerrotaan yleisesti parviälyn käsitteistä ja joistakin sen menetelmistä. Toisessa luvussa määritetään, mitä optimointiongelma ja (meta)heuristiikka tarkoittavat sekä pohditaan hakuongelman rajoitteita. Lisäksi samaisessa luvussa käsitellään ACO:n ja SDS:n perusteita ja annetaan molemmista tekninen esimerkki; muurahaisoptimointi on varsin hyvä kombinatoristen ongelmien, kuten kauppamatkustajan ongelman likimääräiseen ratkaisuun. Stokastinen diffuusiohaku toimii puolestaan hyvin hahmonsovitukseen liittyvissä ongelmissa.

Kolmannessa luvussa käsitellään parvioptimointia ja sen perusversioon liittyviä käsitteitä, kuten eri parametrien vaikutusta. Lisäksi tarkastellaan eri muotoja, kuten adaptiivisuutta, ja perusversiota tehokkaampia tekniikoita. Monitavoiteoptimointi sivuutetaan liian laajana aiheena, vaikka se on tärkeä PSO:n käytön osa-alue [Kod07]. Lisää parvioptimoinnista on neljännessä luvussa. Siinä käydään läpi useita edistyneitä PSO-tekniikoita, jotka erityisesti pyrkivät toimimaan hyvin myös usean optimiratkaisun tapauksissa. Näistä tekniikoista tässä työssä testattiin CLPSO:ta (Comprehensive Learning Particle Swarm Optimizer)[Lia06], mutta tulosten perusteella se ei suoriutunut kovin hyvin verrattuna esimerkiksi kolmannessa luvussa esiteltävään FIPSiin (Fully Informed Particle Swarm) [Ken05].

Työn tarkoituksena on selvittää PSO:n perusversion, FIPSin ja CLPSO:n eroja, sekä kuinka hyvin rinnakkaisevoluution ja PSO:n yhteismenetelmä onnistuu kehittää tekoälypelaajan Connect-lautapeliin [Sch02]. Työssä arvioidaan myös eri parametrien, kuten haussa käytetyn iteraatiomäärän ja rinnakkaisevoluutiossa käytettävien vastustajien määrän vaikutusta tulokseen. Kehitettyjä tekoälyä käyttäviä pelaajia testattiin kolmea referenssipelaajaa vastaan, jotka olivat satunnaisen siirron tekevä, naiivia logiikkaa sisältävä ja minimax-algoritmia käyttävä pelaaaja.

Saatujen tulosten perusteella kehittäminen onnistuu kohtuullisesti erityisesti FIPS:iä käyttämällä. Lisäksi lisäämällä iteraatiomäärää tekoälyä käyttävä sovellus saadaan kehitettyä keskimäärin selvästi paremmaksi. Esitettäviin tuloksiin on kuitenkin syytä suhtautua varauksella – laskentaresursseja olisi tarvittu melkoisesti enemmän tilastollisesti

merkittävämmän datan luontiin. Lisäksi muilla kuin 3x3-pelilaudalla testit ovat vielä sangen alustavia.

2 Parviälyn perusteita

2.1 Yleistä parviälystä

Älykkyyttä on perinteisesti pidetty yksilön inhimillisenä ominaisuutena. Luonnossa on kuitenkin huomattu ilmiöitä, joiden perusteella älykkyyttä voidaan yhtä hyvin pitää alkeellisempienkin olentojen muodostaman yhteisön ominaisuutena. Esimerkiksi muurahais- ja hyönteisyhteisöissä on kompleksisuutta, jota ei voida selittää pelkästään tarkastelemalla yhteisössä elävän yksilön kykyjä. Myös ihmisten uskomusten ja oppien kehittymisen voidaan nähdä sosiaalisena älykkyytenä. Karkeasti ottaen ihmisen mielipiteeseen vaikuttaa oma aiempi historia sekä sen sosiaalisen yhteisön mielipide, johon ihminen kuuluu. [Ken01]

Seuraavaksi esitetään parviälytekniikoille yhteisiä piirteitä; huomiot pohjautuvat pääosin Banabeun, Dorigon ja Theraulazin oppikirjaan. [Bon99]

Jos yleistetään hieman, voidaan parviälystä tehdä seuraavia huomioita: Parviälymenetelmien ideat ovat lähtöisin pääosin luonnosta. Yhteistä niille on, että ongelmanratkaisussa käytetään joukkoa yksinkertaisia agentteja. Agenttien mahdolliset toimenpiteet ovat hyvin rajattuja eikä niillä ole kehittynyttä älyä; ne eivät siis vastaa varsinaisesti Norvigin ”AI: Modern Approach” –kirjassa esitettyjä älykkäitä agentteja. [Nor03] Parviälyagentit toimivat satunnaisuutta hyödyntäen ainoastaan lokaalin tiedon perusteella. Ei siis ole olemassa keskitettyä älyä, joka ohjaisi agentteja muodostamaan osaratkaisuista lopputulosta. Ratkaisu kumpuaa (engl. emerging) agenttien näennäisen epämääräisestä toiminnasta *itseorganisoidusti*. Itseorganisoituvuusominaisuus on merkittävä mm. siksi, että tekoäly voidaan hajauttaa itsenäisesti toimiviin yksiköihin. Toisaalta yksilöiden toiminnallisuusjoukon määrittäminen halutun kokonaisuuden aikaansaamiseksi on vaikeaa kaaosmaisen syy–seuraus-suhteen vuoksi. [Bon99]

Muita tärkeitä useisiin parviälymenetelmiin liittyviä ominaisuuksia ovat nk. *stigmergy*[1], positiivisen palautteen kierre ja satunnaisuus. S*tigmergy* tarkoittaa, että agentit viestivät muokkaamalla ympäristöä. Se yhdessä itseorganisoituvuuden kanssa mahdollistaa

---

[1] Stigmergy on muodostettu kahdesta kreikankielisestä sanasta: stigma ’pistos’ ja ergon ’työ’. [Bon99]

esimerkiksi halpojen robottien rakentamisen. Tämä seuraa siitä, että robotteihin ei tarvitse erikseen lisätä radioita, mikä komponenttikustannusten lisäksi rajoittaisi robottimäärän skaalautuvuutta. Luonnossa tapahtuvasta ympäristön muokkaamisesta käyvät esimerkkinä sangen mutkikkaan muotoisen pesänsä rakentavat termiitit, joiden työ etenee ympäristön virikkeisiin reagoimalla. [Bon99]

Optimoinnin kannalta merkittävä esimerkki *stigmergyn* toiminnasta ovat feromonit. Feromoni on kemikaali, jota useat eläinlajit käyttävät viestimiseen. Sen avulla voidaan esimerkiksi vaikuttaa agentin toimenpiteen todennäköisyyteen. Muurahaisyhteisöissä feromonijäljen pitoisuus kertoo, kuinka paljon polkua pitkin on mennyt muurahaisia lähimenneisyydessä. Mitä enemmän pitoisuutta reitillä, sitä voimakkaammin muurahaiset haluavat sitä kautta mennä. Lisäksi reiteiltään pesään palaavat muurahaiset kasvattavat lyhyimmän polun feromonipitoisuutta nopeimmin, sillä lyhyimmän reitin valitsevat muurahaiset palaavat nopeiten takaisin. Näin kasvaa todennäköisyys sille, että lyhyin polku valitaan myös seuraavaksi. Tämä ilmiötä kutsutaan positiivisen palautteen kierteeksi tai *autokatalysmiksi*, ja se on elinehto myös muissa luonnonyhteisöiden ilmiöissä. [Bon99]

Kaikissa parviälyn menetelmissä keskeisimpiä ominaisuuksia on satunnaisuus, sillä se mahdollistaa uusien ratkaisujen löytämisen sekä tarpeeksi laajan ja hajautuneen ehdokasratkaisujoukon. Esimerkiksi muurahainen saattaa eksyessään ravinnonkeruumatkallaan löytää uuden ravintolähteen, joka voi jopa olla lähempänä pesää. Feromonijäljen jättämisen ansiosta vähitellen tieto paikasta leviää muillekin. [Bon99]

2.2 Parviälymenetelmiä

Vaikka tämän työn keskeisin teema on parviälyn metaheuristiikat, on luonnon järjestelmistä löydettävissä muita tekoälynäkökulmasta kiinnostavia ilmiöitä, joista on syytä mainita muutama lyhyesti. Vaikka osa niistä soveltuukin vain rajattuihin tehtäviin, niillä on erityistä mielenkiintoa robotiikan tutkimuksessa. Robotiikassa hajautetulla itseorganisoituvalla mallilla on useita hyviä puolia, kuten edulliset kustannukset ja ulkopuolisen kontrollin tarpeettomuus. Esimerkiksi tulevaisuudessa nanoteknologia voi mahdollistaa, että ihmisen kehoon ujutetaan vahingottuneita soluja korjaava robottiparvi. [Bon99]

Seuraaviksi esiteltävät menetelmät ovat esitelty Dorigon ja Stützlen oppikirjassa [Dor04]. Muurahaisten ravinnonhankintaan (engl. foraging) perustuvasta menetelmästä kerrotaan sen sijaan luvussa 3.3.1.

- Useissa muurahaisyhteisöissä muurahaiset kokoavat pieniä munia ja toukkia keskelle pesän jälkikasvualuetta sekä siirtävät suurimman toukan jälkikasvurykelmän reunalle. Tämän ilmiön, jälkikasvulajittelun (engl. brood sorting), mukaisessa stokastisessa mallissa muurahainen mahdollisesti ottaa maassa olevan jälkikasvun mukaansa tai lopettaa jälkikasvun siirtämisen. Todennäköisyys valinnalle määräytyy ko. jälkikasvutyypin määrästä lähiympäristössä; paikalleen jättäminen on todennäköistä silloin, kun samantyyppisiä on paljon ja siirtäminen silloin, kun vieressä ei ole juuri samanlaisia. Tämän mallin mukaisesti erivaiheiseessa olevat jälkeläiset kerääntyvät vähitellen omiin kasoihin. Ideaa voidaan soveltaa erilaisiin lajitteluongelmiin, joissa joukko eri tyyppisiä esineitä halutaan siirtää omiin joukkioihinsa. Toimivuutta on kokeiltu kehittämällä robotteja, jotka ryhmittävät esineitä kasoihin ilman minkäänlaista keskitettyä ohjausta. Menetelmän tärkeä piirre on, että esineiden alkusijoittelu määrää agenttien toiminnan. [Dor04]

- Muuraisyhteisöissä yksilöt erikoistuvat usein joihinkin tiettyihin tehtäviin (engl. division of labor). Ne pystyvät kuitenkin toimimaan myös muissa tehtävissä silloin, kun jokin toimenpide on poikkeuksellisen kiireellinen; tällöin toimenpiteen virikemäärä (engl. stimuli) ylittää yksilön herkkyysrajan. Tällaista joustavuutta tarvitaan ongelmissa, jotka vaativat jatkuvaa soveltumista muuttuvaan tilanteeseen. Menetelmää voidaan käyttää optimoimaan yksilöiden toimintaa siten, että yksilön herkkyysraja on korkea poikkeuksisille tai kustannukseltaan kalliille toimenpiteille. Tällöin normaalioloissa yksilöt eivät käytä epäedullista toimenpidettä usein. [Dor04]

- Kun muurahaiset huomaavat saaliinsa olevan liian raskas yksinkuljetettavaksi, ne värväävät apuvoimia, sillä muurahaiset osaavat kuljettaa suurenkin saaliin tehokkaasti ryhmänä (engl. cooperative transport). Ne kääntyvät ja sijoittuvat uudestaan aina tarvittaessa niin, että kuljettaminen etenee kohti pesää. Ilmiötä

mallintavaa käyttäymistä on kokeiltu onnistuneesti roboteilla, joilla työnnetään laatikkoa maalia kohti. [Dor04]

Edellä mainittujen lisäksi on useita muita parviälymenetelmiä, jotka sivuutetaan tässä tutkielmassa. Niistä mainittakoon *Cooperative Hunters* ja *Cooperative Cleaners*. *Cooperative Huntersissa* joukko miehittämättömiä lentoaluksia (UAV) pyrkii löytämään liikkuvat kohteet, joiden alkusijaintia ei tiedetä. Kohteet sen sijaan tietävät lentoaluksien sijainnin koko ajan ja osaavat väistellä niitä älykkäästi. *Cooperative Cleanersissa* puolestaan parvi yksilöitä pyrkii puhdistamaan annetun graafin likaiset solmut mahdollisimman nopeasti, ilman että he käyttävät erillistä viestintäyhteyttä. [Alt06]

3 Parviälymetaheuristiikkoja

3.1 Optimointiongelma

Optimointiongelma on yleinen tietojenkäsittelytehtävä, joka voidaan määritellä seuraavasti. On olemassa algoritminen ongelma, jolle halutaan löytää paras mahdollinen ratkaisu. Ratkaisua varten ongelmasta luodaan abstrakti malli, joka määrittää ongelman formaalisti. Erityisesti määritetään hakuavaruus, joka koostuu kaikista niistä mahdollisista vaihtoehdoista, jotka kelpaavat ongelman ratkaisuksi. On tärkeää huomata, että malli voidaan esittää monella tavalla. Niinpä mallin muoto vaikuttaa siihen, kuinka ratkaisu voidaan muodostaa ja kuinka tehokas se voi olla. [Mic00]

Hakuavaruus ei kuitenkaan yksistään riitä; tarvitaan keino tietää, mitkä vaihtoehdoista ovat ratkaisuja. Tätä varten esitetään *tavoitefunktio* (engl. objective function), joka on ratkaisua formaalisti kuvaava lauseke. [Mic00] Esimerkiksi tavoitteena voisi olla löytää lausekkeen $x \cdot x$ minimoiva x, joka kuuluu joukkoon S. Tämä esitettäisiin tavoitefunktiona muodossa:

$$\min_{x \in S} x \cdot x \ .$$

Hakuavaruus ja tavoitefunktio riittävät ongelman ratkaisemiseen, mutta vain silloin, kun algoritmin suoritusajalla ei ole merkitystä. Kun hakuun on käytettävissä rajattu aika, ei voida olla varmoja, että kaikki ehdokkaat ehditään tutkimaan. Tällöin voidaan tyytyä johonkin ratkaisuehdokkaaseen, joka ei ole tavoitefunktion mukainen, mutta tarpeeksi lähellä sitä. Ongelmana kuitenkin on, että haku ei voi verrata ratkaisuehdokkaiden hyvyyttä toisiin ratkaisuihin. Näin ei myöskään voida ohjata hakua lupaavaan suuntaan hyväksikäyttäen hakuavaruuden rakennetta. Tätä varten tarvitaan *evaluointifunktio* (engl. evaluation function), joka antaa argumenttina saadun ratkaisuehdokkaan hyvyyden. Evaluointifunktio on siis mittari, jonka avulla hakuavaruuden osia voidaan verrata toisiinsa. Esimerkkinä ratkaisuehdokkaiden evaluoitu arvo voisi olla arvoltaan väliltä yhdestä kymmeneen, kun kymmenen on teoreettinen maksimiraja ratkaisuehdokkaan hyvyydelle. [Mic00]

Evaluointiofunktion ja tavoitefunktion välillä on sidos: evaluointifunktion täytyy antaa paras (minimi tai maksimi) arvo tavoitefunktion mukaiselle ratkaisulle, jota kutsutaan globaaliksi ratkaisuksi tai optimiksi. Evaluointifunktio ei kuitenkaan ole välttämättä helppo muodostaa. Joskus sen voi suoraviivaisesti johtaa tavoitefunktiosta, mutta tätä ei suinkaan aina voida tehdä. Lisäksi on mahdollista, että vain osa hakuavaruudesta antaa järkeviä ratkaisuja; evaluointifunktio onkin syytä muodostaa ongelman yksityiskohtia hyödyntäen, jotta hakeminen olisi mahdollisimman tehokasta. [Mic00] On syytä huomauttaa, että useissa julkaisuissa käytetään pelkästään termiä tavoitefunktio, vaikka tarkoitetaan evaluointifunktiota.

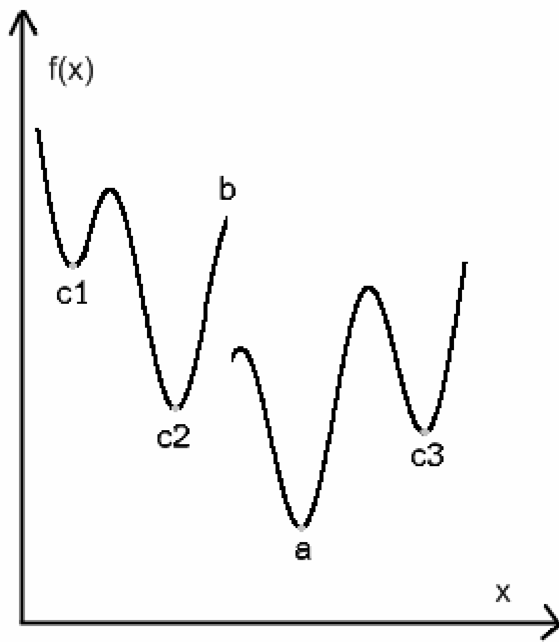
Optimointi- eli hakuongelma voidaan esittää nyt evaluointifunktion avulla formaalisti.

```
Etsi ratkaisuehdokas x, joka toteuttaa evaluointifunktiolle f:
f(x) ≤ f(y), missä y on mikä tahansa ratkaisuehdokas.
```

Ylläolevan mukainen hakuongelma on minimointiongelma. Maksimointiongelmaksi se saadaan vaihtamalla $\leq$:n tilalle $\geq$. [Mic00]

Optimoitava evaluointifunktio voi olla diskreetti tai jatkuva sekä sisältää epäjatkuvuuskohtia. Se voi olla myös *multimodaalinen* eli sisältää useita optimaalisia kohtia hakuavaruudessa. Lisäksi pisteitä, joiden evaluoitu arvo on suurempi kuin minkään muun lähiympäristössä olevan pisteen, kutsutaan lokaaliksi tai paikalliseksi optimiksi. [Ken01]

Kuva 1 antaa esimerkin eräästä minimoitavasta funktiosta.



Kuva 1. Minimoitava funktio graafisena esityksenä. *x*-akseli ilmoittaa hakuavaruuden paikan ja *f(x)* evaluointifunktion arvon pisteessä *x*. *a* on globaali minimi, *b*:ssä on epäjatkuvuuskohta ja *c1*, *c2* ja *c3* ovat *f*:n paikallisia optimeja (minimikohtia).

On myös huomionarvoista, että tietojenkäsittelyssä tärkeät diskreetit rajoiteongelmat voidaan muuntaa minimoitavaksi funktioksi. Esimerkiksi boolen kaavan toteutumisongelmassa (SAT) pyritään löytämään arvotus, jolla tietty boolen kaava saadaan voimaan. [Kle06] Optimointiongelmaksi se saadaan muutettua, kun minimoidaan ei-toteutuvien klausuulien määrää.

3.2 Heuristiikat

3.2.1 Yleistä

Käytännössä ongelman tarkan optimaalisen arvon hakeminen on useasti mahdotonta, sillä hakuavaruus on niin suuri. Käytettävissä olevat resurssit eivät ehkä tällöin riitä, vaikka ei käytäisi läpi kaikkia vaihtoehtoja; ahneen algoritmin, hajoita- ja hallitse -menetelmän tai dynaamisen ohjelmoinnin käyttäminen haun tehostamiseksi ei myöskään välttämättä auta. Niinpä usein joudutaan tyytymään etsimään riittävän hyviä arvoja. [Dor04]

Algoritmia, joka etsii hyvää, mutta ei välttämättä optimaalista ratkaisua erilaisia sääntöjä soveltamalla kutsutaan *heuristiikaksi*. Heuristiikkoja kehitetään usein tarkastelemalla jonkin

tietyn ongelman erikoisominaisuuksia. Niinpä ne ovat valitettavasti usein sovellusaluekohtaisia eivätkä ole hyödynnettävissä muissa ongelmissa. [Dor04]

Heuristiikkoja on kahta erilaista tyyppiä: konstruktiivisia tai paikallista hakua soveltavia. Konstruktiiviset heuristiikat aloittavat tyhjästä ja osa kerrallaan muodostavat ratkaisuehdokasta. Esimerkiksi SAT-ongelmassa konstruktiivinen algoritmi voisi valita ensin jollekin muuttujalle totuusarvon, tarkastaa klausuulien toteutumista ja valita uudelle muuttujalle totuusarvon. Algoritmi jatkaisi toimintaansa vastaavalla tavalla, kunnes kaikille muuttujille olisi valittu arvo ja ratkaisuehdokas olisi muodostettu. Paikallinen haku puolestaan aloittaa valmiista ratkaisuehdokkaasta ja pyrkii löytämään paremman ratkaisun tekemällä paikallisia muutoksia ratkaisuehdokkaaseen. Esimerkiksi SATissa jokin lokaali haku voisi valita toteutumattoman klausuulin ja vaihtaa muuttujien arvoja niin, että klausuuli toteutuu. Toisaalta vaihdon seurauksena voi jonkin muun klausuulin totuusarvo vaihtua todesta epätodeksi. Lokaali haku ei siis pysty hahmottamaan kokonaistilannetta, ja sen vuoksi se jääkin usein lokaaliin optimiin. Ratkaisuna usein suoritetaan lokaalihaku monta kertaa, eri alkulähtökohdista. [Dor04]

On kuitenkin huomattu, että on olemassa yleiskäyttöisiä heuristiikoita, metaheuristiikkoja. Metaheuristiikka voidaan nähdä algoritmikehyksenä, jota voidaan soveltaa eri ongelmille määrittämällä parametrit ongelmaan sopiviksi. Ongelmaa varten voi myös joutua määrittämään, miten jokin korkealla tasolla kuvattu heuristiikan osa pitää tulkita juuri ko. ongelmassa. Esimerkiksi *Laske yhteen* -niminen proseduuri voisi vektoriavaruudessa laskea vektoreiden summan, kun taas sama operaatio voisi merkkijonotehtävässä yhdistää kaksi merkkijonoa peräkkäin. Metaheuristiikka voi myös hallita usean aliheuristiikan toimintaa, ohjata niitä oikeaan suuntaan ja yhdistää niiden tuloksia. Tällä tavoin vältytään juuttumiselta paikallisiin optimeihin. [Dor04]

Perinteisiä metaheuristiikoita, jotka parantavat ratkaisuehdokasta iteraatio kerrallaan, ovat mm. simuloitu jäähdytys (engl. simulated annealing), tabuhaku ja kukkulalle kiipeäminen (engl. hill climbing). [Mic00] Evoluutiopohjaisissa optimointiheuristiikoissa sen sijaan pidetään joukkoa ratkaisuehdokkaita, geenejä, joiksi ongelman ratkaisuehdokkaat on koodattu. Geenejä mutatoidaan satunnaisesti ja risteytetään keskenään luoden uusia sukupolvia käyttäen ehdokkaan kelpoisuutta (engl. fitness) määräämään, millä todennäköisyydellä geeni valitaan seuraavan sukupolven jälkeläisen vanhemmaksi.

Säilyttämällä useita, mahdollisesti hyvinkin erilaisia ratkaisuehdokkaita saavutetaan se etu, että huonollakin ehdokkaalla on pieni mahdollisuus tulla mukaan seuraavaan sukupolveen – kenties sitä mutatoimalla tai risteyttämällä toiseen ehdokkaaseen löytyy uusi, parempi ratkaisu. [Ken01] Evoluutiopohjaisten metodien menestys on hyvä esimerkki luonnossa esiintyvien ilmiöiden potentiaalista tietojenkäsittelyongelmien ratkaisemisessa.

3.2.2 Satunnaisuus

Satunnaisuus on optimoinnissa tärkeää monesta syystä. Sen ansiosta voidaan usein käyttää käsitteellisesti hyvin yksinkertaista menetelmää, koska haun historiaa ei tarvitse säilyttää muistissa erikseen. Erityisesti toiminnaltaan hajautetuissa, eri osapuolien välistä viestintää sisältävissä algoritmeissa voidaan vähentää kahden toisensa tuntevan osapuolen välistä suoraa viestintää. Satunnaisuus auttaa myös estämään algoritmin lukkiutumista yhteen tilaan. [Kle06] Selvästi nämä pätevät myös parviälyssä, sillä tässä tapauksessa viestintämäärä on usein hyvin pientä ja tapahtuu epäsuorasti, eikä agenttien sisäinen muistimäärä tarvitse olla suuri. Lisäksi satunnaisuus ja usean ratkaisun yhtäaikainen ylläpitäminen estävät juuttumista yhteen (alioptimaaliseen) tilaan. [Bon99]

Koska satunnaisuus on tärkeä parviälyn tekijä, on selvää, että mahdollisesti käytettävät todennäköisyysjakaumat on valittava huolella. Usein kuitenkin unohdetaan satunnaislukugeneraattorin merkitys algoritmin toiminnassa. [Cl06a]

Tässä tutkielmassa pseudosatunnaislukugeneraattorilla tarkoitetaan ohjelmallista mustaa laatikkoa, joka alustetaan siemenluvulla ja joka generoi pseudosatunnaisen lukujonon. On oleellista, että uudestaan samalla siemenluvulla alustettu generaattori antaa saman lukujonon. Tarkoituksena generaattorilla on antaa mahdollisimman vaikeasti ennustettavissa oleva lukujono, missä jokaisen lukujonon indeksin kohdalla on mahdollista olla mikä tahansa luku annetulla todennäköisyysmallilla. Tosin tietokoneen lukutarkkuusrajojen vuoksi lukuväliltä voidaan ottaa vain äärellinen määrä lukuja. [Cl06a] Usein luku saadaan tasaisesta jakaumasta $U[\min, \max]$ (usein $U[0, 1]$) tai normaalijakaumasta $N[\mu,\sigma^2]$ (merkitään myös G:llä Gaussin mukaan), missä μ on jakauman odotusarvo ja σ^2 varianssi.

Satunnaislukugeneraattorien toteutuksessa on kuitenkin useimmissa tapauksissa merkittäviä puutteita. Esimerkiksi C-ohjelmointikielen standardikirjaston generaattori ei pysty antamaan itseisarvoltaan tarpeeksi pieniä arvoja. Clercin kokeiden mukaan generaattorin

laadun vaikutus oli huomattava. Onkin syytä tutkia generaattorin laadukkuutta ennen sen käyttöä satunnaisalgoritmeissa. Clerc ehdottaa käytettäväksi KISS-generaattoria, jonka käyttöoikeudet ovat vapaat. [Cl06a]

3.2.3 Heuristiikkojen rajoista

(Meta)heuristiikkojen tutkimisella pyritään löytämään mahdollisimman tehokas yleiskäyttöinen heuristiikka. Ei kuitenkaan ole selvää, voiko sellaista ylipäänsä tehdä. Nimittäin ”Ei ilmaista lounasta” -teoreema (engl. No Free Lunch, NFL) väittää, että ei ole olemassa heuristiikkaa, joka olisi yleisesti parempi kuin mikä tahansa muu haku, jos otetaan huomioon kaikki mahdolliset ongelmat. Väite on intuitiivisesti selvä, sillä jos funktion rakenne on mielivaltainen, ei heuristiikasta ole apua optimin löytämisessä kuin harvoin. [Wol96] Oleellinen kysymys onkin, ovatko useimmat (älylliselle oliolle) merkitykselliset optimointiongelmat jollakin tapaa poikkeavia. Jos näin on, kenties jokin heuristiikka voi suoriutua niissä paremmin. Silloin ei olisi väliä, vaikka ko. heuristiikka ei suoriutuisi toisenlaisissa ongelmissa ollenkaan. Onkin tutkittu, mikä mahdollistaa tehokkaan funktion optimin haun [Chr01].

Yleisesti ottaen metaheuristiikat pyrkivät hyödyntämään funktion säännönmukaisuuksia. Eräs tällainen ominaisuus on itsesamankaltaisuus (engl. self-similarity): hakuavaruudessa lähekkäin olevilla pisteillä on samankaltaisia arvoja [Chr01]. Hakuavaruuden muoto on siis heuristiikan elinehto, mutta toisaalta myös haaste. Erityisesti paikalliset optimit aiheuttavat ongelmia heuristiikoille, sillä lokaalista optimista pois päästäkseen haun täytyy tehdä voimakasta vastustusta lähiympäristön informaatiota kohtaan. Heuristiikkoilla täytyy siis olla jokin selvä menetelmä välttääkseen juuttumista ei-globaaliin optimiin. Simuloitu jäähdytys -heuristiikassa haulle annetaan mahdollisuus siirtyä kohti huonolta vaikuttavaa paikkaa todennäköisyydellä, joka riippuu siitä, kuinka kauan haku on kestänyt. Tabuhaussa puolestaan ylläpidetään tietoa paikoista, joissa ollaan jo käyty. Populaatiopohjaisissa heuristiikoissa sen sijaan ratkaisuehdokasjoukkoon voidaan sisällyttää pienellä todennäköisyydellä huonojakin ratkaisuja ympäri hakuavaruutta. [Mic00]

3.2.4 Heuristiikkojen testaus

Heuristiikkojen vertailua varten tarvitaan hyviä, toisistaan poikkeavia testiongelmia. Vertailu voidaan suorittaa mm. iteraatioiden tai evaluointifunktion käyttömäärän suhteen, jos asetetaan jokin raja, milloin löydetty tulos on tarpeeksi hyvä. Ongelmissa, joissa hyvän globaalin optimin saavuttaminen on vaikeaa, voidaan verrata heuristiikkojen parhaita

arvoja. Testejä pitää tehdä lukuisia kertoja, jotta vertailu olisi tilastollisesti kelvollinen. Voidaan esimerkiksi testata, kumpi annetuista heuristiikoista on parempi valitussa ongelmassa, ja antaa tulokselle luotettavuusarvo.

Seuraavassa on esimerkkejä usein käytetyistä testifunktioista, joilla voidaan testata heuristiikkojen optiminhakuhyvyyttä. Funktiolistassa on ensin annettu yleisesti käytetty nimi, minkä jälkeen on funktion analyyttinen muoto ja lopuksi arvoalue kullekin ulottuvuudelle. [Cl06a]

`Alpine` $\sum_{d=1}^{10} \left| x_d \cdot \sin(x_d) + 0{,}1 x_d \right|,\ x_d \in [-10,10]$

`Parabola` $\sum_{d=1}^{30} x_d^{\ 2},\ x_d \in [-20,20]$

`Rosenbrock` $\sum_{d=1}^{29} (1 - x_d)^2 + 100(x_d^{\ 2} - x_{d+1})^2,\ x_d \in [-10,10]$

[Cl06a]

3.3 Muurahaisoptimointi

3.3.1 Perusidea

Muurahaisoptimointialgoritmien malli on lähtöisin luonnossa elävien muurahaisten ravinnonetsintäkäyttäytymisestä. Kun muurahainen saapuu ”tienristeykseen”, sen valitsema reitti määräytyy stokastisesti valittavissa olevien teiden feromonipitoisuudesta. Lisäksi, koska muurahainen liikkuessaan jättää itse lisää feromonia, lyhyin reitti alkaa kasvattaa pitoisuuttaan. Näin ko. reitin todennäköisyys tulla valituksi jatkossa kasvaa. On myös tärkeää, että feromonit haihtuvat ajan kuluessa. Se mahdollistaa uuden suotuisamman polun valinnan, jos vanha ruokavaranto ehtyy tai löydetään uusi parempi. Vaikka luonnossa haihtuminen on usein hyvin hidasta, voidaan tietojenkäsittelyongelmissa haihtumisnopeus valita rajoituksetta sopivaksi. [Bon99]

Ensimmäinen muurahaisyhteisöpohjainen hajautettu laskenta (engl. Ant System) esiteltiin vuonna 1992 [Col92]. Tässä työssä se sivuutetaan; sen sijaan tarkastellaan nk. ACO-metaheuristiikkaa. Lisäksi sivuutetaan muut muurahaisalgoritmit, esimerkiksi graafin minimikustannuspolun etsimiseen käytettävä S-ACO [Dor04].

3.3.2 ACO-metaheuristiikka

ACO-metaheuristiikassa (engl. Ant Colony Optimization) muurahaiset rakentavat ratkaisua vaihe kerrallaan stokastisesti. Sitä voidaan soveltaa mihin tahansa kombinatoriseeen ongelmaan, jolle voidaan määrittää konstruktiivinen heuristiikka. Haasteena on, kuinka esittää optimointiongelma muurahaisoptimoinnin käsitteiden avulla. [Dor04]

ACO:ssa perusajatuksena on antaa muurahaisten kulkea täysin kytketyssä graafissa satunnaisesti, niin että tarpeeksi usean kävelykierroksen jälkeen on saatu aikaiseksi riittävän hyvä ratkaisu. Ratkaisua varten graafin solmuilla (paikat) ja kaarilla (reitit) on feromonijälki τ sekä heuristinen arvo η. Muurahaiset päivittävät feromonijälkeä, jotta tieto muurahaisten hakuprosessin etenemisestä säilyy ja välittyy muille muurahaisille. Heuristista informaatiota puolestaan ei saada muurahaisilta, vaan se on ongelmatapaukseen liittyvää tietämystä. Se voidaan tietää joko ennen hakuprosessin käynnistämistä ongelman yksityiskohtien perusteella tai sitä voidaan saada hakuprosessin aikana ongelman mahdollisesti muuttuessa. [Dor04]

Seuraavassa on lueteltu tarkemmin yhteisöön kuuluvan muurahaisen ominaisuuksia. Tiedot perustuvat Dorigon ja Stützlen kirjaan [Dor04].

- Muurahainen käyttää muistia tallentamaan polun, jonka se on kerennyt kulkea. Muistia voidaan hyödyntää laillisten ratkaisujen muodostamisessa, heuristisen ja löydetyn ratkaisupolun arvon laskemisessa sekä siinä, että osataan kulkea reitti takaisin käänteisesti.

- Muurahainen on aina jossakin tilassa, joka määräytyy käydyistä solmuista. Jos yksikin määritetyistä lopetusehdoista on voimassa, muurahaisen suoritus päättyy ko. kierroksella. Sen sijaan, jos mikään lopetusehto ei ole voimassa, muurahainen siirtyy graafissa naapurisolmuun stokastisen siirtosäännön ohjaamana. Siirtosäännön eri vaihtoehtojen todennäköisyydet muodostuvat paikallisesta feromonijälkitiedosta ja heuristisista arvoista, muurahaisen sisäisen muistin tilasta sekä ongelmakohtaisista rajoituksista.

- Lisätessään uuden solmun tilaansa muurahainen saa päivittää sen solmun ja kaareen liittyvää feromonipitoisuutta. Samoin, kun muurahainen on saanut valmiiksi

ratkaisun, se palaa muistissa olevaa polkua pitkin ja päivittää käytettyjä solmuja halutessaan.

ACO:n korkean tason toiminnan kuvaus voidaan esittää kolmella proseduurilla:

1) Muurahaiset kulkevat aiemmin kerrotulla tavalla graafissa ja muodostavat reittejä eli ongelman ratkaisuehdokkaita. Tämä tapahtuu synkronoimattomasti rinnakkain muiden muurahaisten kanssa.
2) Muurahaisten ratkaisupoluille oleviin solmuihin ja kaariin lisätään feromonia ratkaisun hyvyyden mukaan. Lisäksi feromonia haihdutetaan tasaisesti pois, jotta vanhoja ratkaisuja voidaan unohtaa.
3) Kolmas proseduuri on valinnainen; siinä yleensä suoritetaan toiminnot, joita yksittäinen muurahainen ei voi tehdä. Esimerkiksi käytetään paikallista hakua parantamaan suorituskykyä tai selvitetään viime kierroksella parhaiten suoriutunut muurahainen ja lisätään sen polulle ylimääräistä feromonia

Yllämainittuja kolmea proseduuria suoritetaan haluttu määrä kertoja. Kullekin metaheuristiikan variaation kehittäjälle jää vapaudeksi päättää, suoritetaanko proseduurit vapaasti rinnakkain, vai täytyykö eri niiden välillä olla synkronointia, jotta suoritus etenee toivotulla tavalla. [Dor04]

3.3.3 ACO:n soveltaminen kauppamatkustajan ongelmaan

Koska kauppamatkustajan ongelma (engl. Traveling Salesman Problem, TSP) on yleisesti tunnettu ja runsaasti tutkittu kombinatorinen NP-täydellinen ongelma, on sillä hyvä havainnollistaa ACO:n soveltamista – etenkin koska TSP on helppo esittää ACO:n käsitteillä. [Dor04]

Kauppamatkustajan ongelma on tiivistetysti seuraavanlainen: Kauppamatkustaja haluaa käydä jokaisessa kaupungissa täsmälleen kerran ja palata lopulta lähtökaupunkiin. Ongelmana on löytää halvin (esim. lyhyin) tällainen reitti, kun tunnetaan kaupunkien välillä tapahtuvien siirtymien kustannukset (esim. etäisyys). [Dor04]

Dorigo ja Stützle esittävät yhden mahdollisen tavan kuvata kauppamatkustajan ongelma ACO:lla nimeltä TSP-ACS: Solmut ovat kaupunkeja ja kaaret kaupunkien välisiä reittejä;

heuristinen arvo η on kaarikustannuksen käänteisluku. Ts. mitä pienempi kustannus, sitä todennäköisemmin muurahainen valitsee ko. reitin; feromonijälki $\tau_{i,j}$ kertoo, kuinka haluttua on siirtyä kaupungista *i* kaupunkiin *j* opitun tiedon perusteella. Osaratkaisut saadaan laittamalla jokainen muurahainen kierroksen alussa satunnaiseen kaupunkiin, minkä jälkeen muurahainen lisää kulkiessaan uuden kaupungin ratkaisuunsa, kunnes kaikissa kaupungeissa on vierailtu. Jotta jokaisessa kaupungissa käytäisiin vain kerran, muurahainen muistaa kaupungit, joissa se on jo käynyt. Seuraavaksi määritellään TSP-ACS:n säännöt siirtymiselle ja päivityksille. [Dor04]

Siirtymissääntö määrää, millä todennäköisyydellä muurahainen siirtyy *i* kaupungista *j*:hin ajanhetkellä *t*. ACS-TSP:ssä muurahainen ottaa valinnassa aluksi huomioon vain parametrina annetun määrän *cl* verran lähimpiä kaupunkeja silloin, kun kaupunkeja on paljon. Näin saadaan rajoitettua vaihtoehtojen määrä järkeväksi. [Bon99]

Kun muurahainen on käynyt kaikissa ehdokaslistan kaupungeissa, valitaan lähin vierailematon kaupunki. Siirtymäsääntö *cl*:lle lähimmälle kaupungille on määritelty seuraavasti:

$$j = \begin{cases} \max \arg_{u \in J_i^k} \{\tau_{i,u}(t) \cdot [\eta_{i,u}]^\beta\} & , \; jos\, q \le q_0 \\ J & , \; jos\, q > q_0 \end{cases}$$

missä *q* on *U*[0, 1]-jakaumasta satunnaisesti otettu luku, $0 \le q_0 \le 1$ ja *J* on satunnaisesti valittu kaupunki $p^k_{i,j}(t)$:n mukaan, kun p^k kertoo todennäköisyyden, että muurahainen *k* valitsee kaupungissa *i* kaupungin *j*:

$$p^k{}_{ij}(t) = \frac{[\tau_{i,j}(t)]^\alpha \cdot [\eta_{i,j}]^\beta}{\sum_{l \in J^k{}_i} [\tau_{i,l}(t)]^\alpha \cdot [\eta_{i,l}]^\beta},$$

missä α ja β ovat säädettäviä parametreja, joilla vaikutetaan hakuhistorian (feromoni) ja ennaltatiedetyn heuristisen arvon (kaupunkien etäisyys) vaikutusta kaupungin valintaan;

liika painotus toisen suhteen johtaa joko ennenaikaiseen tietylle reitille juuttumiseen tai liian innokkaaseen läheisten kaupunkien käyttöön. [Bon99]

Feromonimääränpäivityssäännöt poikkeavat ACS-TSP:ssä luvun alussa esitetystä luonnonmukaisesta mallista. Ainoastaan muurahainen, jolla on muistissa paras siihen asti löydetty polku T^+, saa päivittää feromoneja kierroksen *jälkeen* – nimenomaan T^+:lle kuuluvia kaaria. Tämän tarkoituksena on ohjata hakua hyvään suuntaan. Globaali feromonipäivityssääntö on seuraavanlainen:

$$\tau_{i,j}(t) \leftarrow (1-p) \cdot \tau_{i,j}(t) + \frac{p}{L^+}$$

missä i ja j ovat parhaaseen polkuun T^+ kuuluvat kaaret, p on haihtumisnopeuseen vaikuttava parametri ja L^+ on T^+:n pituus. [Bon99]

Lisäksi ACS-TSP:ssä muurahaisen valittua siirtymisen i:stä j:hin se päivittää feromonipitoisuutta τ lokaalin päivityssäännön mukaan, jotta syntyisi muita ratkaisumahdollisuuksia. Ajatuksena on, että muurahaisen kulkiessa reittiä pitkin se vähentää samalla reitin feromonipitoisuutta, jotta vähän käytetyt reitit saavat paremman mahdollisuuden tulla valituksi. Jos näin ei tehtäisi, muurahaiset alkaisivat suosia yhä useammin tiettyä reittiä, joka hyvin todennäköisesti ei olisi paras mahdollinen.

$$\tau_{i,j}(t) \leftarrow (1-p) \cdot \tau_{i,j}(t) + p \cdot \tau_0$$

$$\tau_0 = \frac{1}{n \cdot L_{n,n}},$$

missä n on kaupunkien määrä ja p sama kuin globaalissa päivityssäännössä; τ_0:n arvo on kokeellisesti todettu hyväksi, siten että $L_{n,n}$ on arvio TSP:n ratkaisun polun pituudesta käyttämällä lähin naapuri -heuristiikkaa. [Bon99]

Ohjelma 1 esittää TSP-ACS-algoritmin hahmotelman:

Ohjelma 1.

```
Alusta kaikille kaarille (i, j): τi, j (0) = τ0
Aseta m muurahaista satunnaisiin kaupunkeihin
Iteroi seuraavat askeleet haluttu määrä kertoja:
  for k = 1 to muurahaisia
    Rakenna kierros Tk (t) käyttäen n-1 kertaa
    seuraavia vaiheita:
    if kandidaattilistassa on kaupunkeja
    then j = siirtymäsäännön mukainen kaupunki
             cl:stä lähimmästä kaupungista
    else valitse lähin kaupunki
    Sovella lokaalia feromonipäivitystä
  end for
  Laske muurahaisten reitit ja päivitä T+ ja sen pituus L+
  Sovella globaalia feromonipäivityssääntöä
  Aseta jokaiselle kaarelle (i, j): τi,j(t+1) = ti,j(t)
```

Iterointien jälkeen ACS-TSP:n tuottama ratkaisu on T^+. [Bon99]

Verratessa muihin muurahaispohjaisiin ratkaisuihin TSP-ACS on erityisen hyvä, koska siinä globaalia feromonipäivitystä sovelletaan ainoastaan parhaaseen reittiin, mikä auttaa hyvien reittien muodostumisessa. Se myös vähentää laskentaresurssien tarvetta. Lisäksi lokaalin päivityssäännön ansiosta muurahaiset eivät juutu tietylle reitille. Sen sijaan ne muodostavat useita ratkaisumahdollisuuksia. [Dor04]

3.4 Stokastinen diffuusiohaku

3.4.1 Yleistä

Toinen tutkielmassa esiteltävä metaheuristiikka on *stokastinen diffuusiohaku* (engl. Stochastic Diffusion Search, SDS). Myös sillä voidaan nähdä olevan kytköksiä luontoon: SDS:n toiminta muistuttaa luonnon valintamekanismeja, erityisesti mehiläisten tanssimisvärväystä mesipaikalle; tanssi houkuttelee uusia mehiläisiä, jotka puolestaan omalla tanssillaan kasvattavat ravintolähteellä käyvien mehiläisten määrää. [DeM04]

Vaikka SDS kehitettiin alun perin hahmontunnistukseen [Bis89], se soveltuu yleisesti ottaen minkä tahansa rajoiteongelman ratkaisemiseen. Tärkeä ero muurahais- ja parvioptimointiin

on se, ettei agentti tiedä, kuinka hyvä absoluuttinen arvo hänen ratkaisullaan on missään vaiheessa. Tämä johtuu siitä, että agentti laskee evaluointifunktion arvon vain osittain yhden iteraation aikana eikä tallenna sitä muistiin. [DeM04]

3.4.2 Perusidea

Seuraavaksi SDS esitetään käyttäen hahmonsovituskäsitteitä sujuvuuden vuoksi. Olennaista on, että hakuavaruus on hajotettavissa osiin, joita kutsutaan mikropiirteiksi. Ideana on löytää hakuavaruudesta paras mikropiirteiden sovitus annetulle kohdehahmolle (engl. target pattern), joka sekin esitetään mikropiirteinä. Kohteen ja hakuavaruuden mikropiirteet eivät välttämättä ole samankaltaisia, vaan niiden välillä on kuvaus. Esimerkiksi etsittäessä kolmiulotteisen mallin mukaista kohdetta valokuvasta voi kohde olla kuvattu vektoreilla, kun taas hakuavaruus on kaksiulotteinen pikselijoukko. [DeM04]

Kohteelle haetaan mahdollisimman hyviä arvoja ratkaisuavaruudesta (engl. solution space), minkä mukaan kohde kuvataan hakuavaruuteen. Ratkaisua etsitään agenteilla, joilla on hypoteesiratkaisu eli jotkin valinnat kohteen mikropiirteille. Hypoteesia testataan evaluoimalla arvoja mikropiirteittäin testifunktiolla, joka on siis haun tavoitefunktio. Erityistä testifunktiossa on se, että mikropiirteistä evaluoidaan vain osa, vieläpä satunnaisesti ja riippumattomasti toisistaan. Laskennan kannalta on tehokasta, jos testifunktio voidaan esittää osiensa summana:

$$f(x) = f_1(x) + ... + f_n(x) \ .$$

[DeM04]

Jos osafunktiot ovat painottuneet laskennalliselta kuormaltaan tasaisesti, laskentaa lienee mahdollista skaalata käyttämällä useita laskentayksiköitä.

Hakuprosessin alussa jokainen agentti sijoitetaan ratkaisuavaruuteen, esimerkiksi satunnaisesti. Sen jälkeen agentit toimivat sovitun iteraatiomäärän ajan. Ensin agenttien hypoteesista evaluoidaan satunnaisesti sovittu määrä ratkaisuavaruuden osia. Agentit, joiden evaluoitu arvo läpäisee testifunktion, asetetaan *aktiiviseksi*, kun taas muut asetetaan *passiiviseksi*. Tämän jälkeen on diffuusiovaihe, jossa agenttien testit läpäisseitä hypoteeseja levitetään: Passiiviset agentit saavat uuden hypoteesin joko kopioimalla satunnaiselta aktiiviselta agentilta tai valitsemalla umpimähkään. Menetelmän seurauksena toimivilta

vaikuttavat hypoteesit kopioidaan yhä useammin, ja vähitellen kaikki agentit päätyvät samaan hypoteesiin, optimiin. [DeM04]

Seuraavassa (ohjelma 2) lyhyt algoritminen kuvaus SDS-prosessista:

Ohjelma 2.

```
Alusta agentit
Iteroi n kertaa
  # Testivaihe
  Kaikille agenteille:
    testaa satunnaisia evaluointifunktion osia
    if testi ok then agentti on aktiivinen
    else agentti on passiivinen
  # Diffuusiovaihe
  Kaikille passiivisille agenteille:
    x = Valitse satunnainen agentti
    if x on aktiivinen then kopioi x:n hypoteesi agentille
    else aseta agentille satunnainen hypoteesi
```

[DeM04]

3.4.3 Esimerkki

Seuraavassa esitetään lyhyehkö esimerkki SDS:n toiminnasta.

Oletetaan neljä ihmistä, joiden iät ja hiusten väri ovat ilmoitettu parimuodossa (12, ruskea), (22, musta), (28, ruskea), (59, harmaa). Tehtävänä on löytää 28-vuotias ruskeahiuksinen ihminen; testifunktio antaa siis hyväksynnän mikropiirteille ikä ja väri, jos ne ovat 28 ja ruskea; jos testattava mikropiirre on jotain muuta, testifunktio hylkää hypoteesin. Seuravassa on mahdollinen SDS:n eteneminen, kun agentteja on kolme (Taulukko 1).

Taulukko 1. SDS:n toimintaesimerkki kolmella agentilla. Suoritus etenee rivi kerrallaan.

Satunnainen agenttien hypoteesien alustus		
1. agentti # 28, ruskea	2. agentti # 59, harmaa	3. agentti # 12, ruskea
Testaa ikää => OK asetetaan aktiiviseksi	Testaa ikää => VÄÄRÄ asetetaan passiiviseksi	Testaa väriä => OK asetetaan aktiiviseksi
	Kopioi 1. agentin hypoteesin	

1. agentti # 28, ruskea	2. agentti # 28, ruskea	3. agentti # 12, ruskea
Testaa väriä => OK asetetaan aktiiviseksi	Testaa ikää => OK asetetaan aktiiviseksi	Testaa ikää => VÄÄRÄ asetetaan passiiviseksi
		Kopioi 2. agentin hypoteesin
1. agentti # 28, ruskea	2. agentti # 28, ruskea	3. agentti # 28, ruskea
Mahdollinen lopetuskriteeri: kaikilla sama hypoteesi		

3.4.4 Teoriaa

SDS soveltuu tilanteisiin, joissa testifunktion evaluoiminen on laskennallisesti raskas operaatio ja evaluointi on mahdollista pilkkoa osiin. Tämä johtuu siitä, että osittainen funktion evaluoiminen keventää haun kokonaislaskentataakkaa huomattavasti. Toisaalta vaatimus osittaisevaluoinnista käytännössä tekee SDS:n soveltamisen osiinhajoamattomien funktioiden optimoinnissa kustannukseltaan kannattamattomaksi, sillä tällöin SDS:llä ei ole toiminnallisesti etua esimerkiksi satunnaishakuun nähden. Samasta syystä SDS ei myöskään sovellu sellaisiin evaluointifunktioihin, joiden arvon määrittäminen on laskennallisesti edullista, sillä osittaisen evaluointilaskennan hyöty on silloin suhteessa mitätön muuhun heuristiikan laskentaan. SDS on varteenotettava metaheuristiikka kuitenkin ongelmissa, joissa optimoitavan funktion muoto ei ohjaa hakua oikeaan ratkaisuun, ts. optimifunktion arvot eri hakuavaruuden pisteissä eivät anna selvää osviittaa, missä globaali optimi on. Tällöin haun tehokkuutta (suhteessa satunnaishakuun) voidaan parantaa ainoastaan osittaisella evaluoinnilla. [DeM04] DeMeyerin mukaan Grech-Cin [Gre95] on esittänyt testimenetelmän, jolla voidaan arvioida, onko ongelma sopiva SDS:llä ratkaistavaksi. [DeM04]

Poiketen muurahaisoptimoinnista SDS:ssä ei rakenneta ratkaisua muokkaamalla ympäristöä, vaan ratkaisu muodostuu agenttien suoran viestinnän kautta tapahtuvasta ryhmittymisestä. SDS soveltuu myös dynaamisiin ongelmiin, joissa optimoitava funktio muuttuu, mutta tällöin SDS:ään joudutaan tekemään pieniä muutoksia. Ensinnäkin hypoteesia ei kopioida täsmälleen, vaan se voi erota jollain tavalla (vrt. mutaatio evoluutionäärisissä menetelmissä). Tällöin liikkuva optimi on mahdollista löytää lähistöltä. Toiseksi aktiivinen agentti *A* valitsee satunnaisen agentin, ja jos sekin on aktiivinen ja molemmilla on sama hypoteesi, niin *A* valitsee satunnaisen hypoteesin. Jälkimmäistä menetelmää kutsutaan kontekstisidonnaiseksi SDS:ksi, ja se on tehokkaampi kuin perus-

SDS. Tämä johtuu siitä, että perusversiossa hyvä hypoteesi vetää puoleensa suuren osan agentteja, vaikka ei olisikaan paras mahdollinen. Täten haku on melko rajoittunutta. [DeM06] Muista SDS:n muunnelmista mainittakoon *salainen optimisti*, jossa agentti ei hylkää huonoa evaluointia ja *erakko*, jossa agentti ei suostu levittämään omaa hyvältä vaikuttavaa hypoteesia muille. [DeM04]

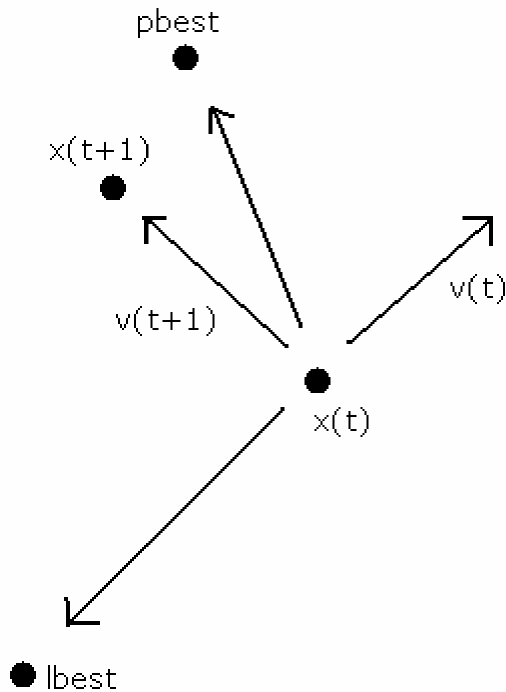
SDS on analysoitu matemaattisesti käyttäen Markovin ketju -teoriaa. On todistettu, että jos haettava kohde (globaali optimi) on hakuavaruudessa, niin agentit myös päätyvät sille jossakin vaiheessa. Jos hakuavaruudessa ei ole kohdetta, niin haku päätyy silti johonkin hypoteesiin – vieläpä testifunktion mukaan parhaaseen. [Nas99]

4 Parvioptimointi (PSO)

4.1 Perusidea

Parvioptimointi (engl. Particle Swarm Optimization, PSO) on metaheuristiikka, joka soveltuu monimutkaisten moniulotteisten funktioiden optimointiin. Sen perusidea on lähtöisin mm. lintuparven liikkeestä. Se kehitettiin toisaalta myös mallintamaan ihmisen sosiaalisvaikutteista oppimista. [Ken95] Pääajatus on, että optimoitavan funktion hakuavaruudessa on joukko erillisiin naapurustoihin jaettuja pisteitä. Niitä kutsutaan agenteiksi tai liikkuvuutensa vuoksi partikkeleiksi. Niiden liikkeeseen vaikuttaa kolme vaikutustekijää: partikkelin oma liikenopeus, oma paras tähän mennessä löydetty sijainti (*pbest*) ja naapuripisteiden paras sijainti (*gbest*) – esitettynä vektorimuodossa. Lisäksi kullekin eri vaikutustekijälle annetaan sen tärkeyden määräävä luottamuskerroin, joka sisältää usein satunnaistekijän. Toisin sanoen partikkelin seuraavan aika-askeleen jälkeinen nopeus ja sijainti määräytyvät sen mukaan, miten partikkeli on itse aiemmin toiminut sekä kuinka naapuruston kautta saadaan tietoa muualla olevista hyvistä pisteistä. [Cl06a] Tässä tutkielmassa käytetään *pbest*- ja *gbest*-termejä tarkoittamaan sekä sijaintia että evaluointifunktion sijainnille antamaa arvoa. Kontekstin perusteella on selvää, tarkoitetaanko termillä sijaintia, arvoa vai molempia.

Kuva 2 havainnollistaa, miten vaikutustekijät nopeusvektori $v(t)$, oma paras (*pbest*) ja naapuruston paras (*lbest*) ajanhetkellä t muuttavat partikkelin nopeuden $v(t+1)$:ksi ja sitä kautta sen vanhaa paikkaa $x(t)$ (mahdollisesti) paremmaksi $x(t+1)$:ksi.



Kuva 2. Partikkelin vaikutusvoimat ja siirtyminen *x*(*t*):stä *x*(*t+1*):seen.

Parvioptimoinnin perusversion ydintoiminta voidaan kuvata seuraavasti (ohjelma 3):

Ohjelma 3.

```
Alusta partikkelit satunnaiseen paikkaan
while iteraatioita jäljellä
  for jokaiselle partikkelille p
    1. for jokaiselle ulottovuudelle i
         vp(i) <-    c1 · vp(i)
                   + c2 · (pbestp(i) – xp(i))
                   + c3 · (lbestp(i) – xp(i))
         xp(i) <- xp(i) + vp(i)
    2. päivitä pbestp/lbestp, jos uusi paikka xp on parempi
```

[Cl06a]

Yllä *v* on nopeus, *pbest* henkilökohtainen paras sijainti ja *lbest* naapuruston paras sijainti. *c1*, *c2* ja *c3* ovat luottamuskertoimet em. vaikutustekijöille; *c2*:n ja *c3*:n arvo lasketaan joka aika-askeleella uudestaan kaavalla; *cMax* · *random*(0, 1), missä *random* antaa satunnaisen luvun jakaumasta *U*[0, 1]. [Cl06a] *c1*:n ja *cMax*:n valinnasta kerrotaan luvussa 4.2.

Koska ulottuvuuksilla on usein määritelty yksinkertainen rajoitus partikkelin sijainnille, pitää erikseen huomioida, mitä tapahtuu, kun partikkelin uusi sijainti menee rajan yli.Yksinkertainen toimiva tapa on asettaa nopeus nollavektoriksi ja partikkelin sijainti ulottuvuuden minimiksi tai maksimiksi sen mukaan kumman reuna-arvon partikkeli ylittää. [Cl06a]

Merkittäviä seikkoja luottamuskertoimien lisäksi on ainakin partikkelien määrä eli parven koko ja naapuruston topologia eli se, kuinka informaatio parhaasta paikasta leviää partikkelien välillä. Myös useat muut parvioptimoinnin perusversiosta kehitettyjen muunnelmien yksityiskohdat vaikuttavat optimoinnin onnistumiseen. [Cl06a] Sen vuoksi seuraavaksi tarkastellaan eri osatekijöiden merkitystä.

4.2 Parametrien arvot perusversiossa

Parven koko eli partikkelien määrä vaikuttaa siihen, kuinka monta kertaa evaluointifunktiota käytetään yhden iteraation aikana. Intuitiivisesti voisi kuvitella, että evaluointimäärää olisi syytä minimoida, jotta hakeminen olisi tehokasta. Toisaalta pitää kuitenkin huomioida, että vähäisellä partikkelimäärällä hakuavaruuden kattavuus ei ole suuri, jolloin optimin löytäminen kestää kauemmin tai hyvää tulosta ei saavuteta ollenkaan. Perusalgoritmille tehtyjen empiiristen kokeiden mukaan parven koko on melko riippumaton muista kertoimista, ja sen ollessa 20:n ja 30:n välillä saadaan hyviä tuloksia perustestijoukolla, jos oletetaan parven koon pysyvän samana koko haun ajan. [Cl06a]

PSO:n luottamuskertoimet *c1*, *c2*, *c3*, joista jälkimmäiset kaksi määräytyvät satunnaisesti *cMax*in mukaan, vaikuttavat oleellisesti haun käyttäytymiseen. Itse asiassa alkuperäinen versio oli määritelty ilman *c1*:stä ja *cMax*:ia, jolloin vaikutusvoimien kerroin oli pelkkä satunnaisluku väliltä [0, 1] kerrottuna kahdella (2). [Ken95] Tällöin ongelmana on esimerkiksi partikkelin edestakainen, kiihtyvä poikkoilu hakuavaruudessa, mitä täytyi vaimentaa käyttämällä nopeusrajoitusparametria *vMax*. [Ken01] Ottamalla *c1* ja *cMax* käyttöön voidaan jättää *vMax*-rajoite pois sekä varmistua siitä, että partikkeli lähestyy lopulta jotakin pistettä eikä oskilloi pitkin hakuavaruutta. [Cl06a]

Luottamuskertoimien määräämiseen on joitain suuntaa antavia ohjeita: *c1*:n eli partikkelin edellisen iteraatiokerran nopeuden kerroin ei haun kannalta ole suotavaa olla ykköstä (1) suurempi. Muutoin tilanteessa, jossa partikkelin paras sijainti on ja pysyy jonkin aikaa naapuruston parhaana, partikkelin nopeus mahdollisesti kertautuu niin, että partikkeli poikkoilee hakuympäristössä eikä lähesty kohti mitään pistettä. Kerroin *c1* voidaan valita myös ykköstä suuremmaksi, mutta silloin joudutaan ottamaan *vMax* käyttöön. Jos taas *c1* on arvoltaan liian pieni, partikkelin vauhti hiipuu liian aikaisin ja pysähtyy huonoon paikkaan. [Cl06a]

c1:sen tilalle on ehdotettu lineaarisesti muuttuvaa kerrointa *w* [Shi99]. Se on haun alussa suurehko ja sitä vähennetään iteraatioiden kasvaessa. *w* on määritelty seuraavasti:

$$w = (w_1 - w_2)\frac{i_{Max} - i}{i_{Max}} + w_2 \text{ ,}$$

missä w_1 on *w*:n arvo alussa ja w_2 lopussa, i meneillään oleva iteraatio ja i_{Max} iteraatioiden korkein mahdollinen määrä. *w*:n käytöllä saadaan usein parannettua PSO:n suorituskykyä merkittävästi. [Rat04]

c1:n ja *c2*:n määräävä *cMax* ei myöskään saa olla kovin suuri. Matemaattisen analyysin perusteella *c1*:stä ja *cMax*:ia ei pidä valita riippumattomasti. Niille on selvitetty ensin kokeellisesti ja myöhemmin matemaattisesti sopivia arvoja; esimerkiksi *c1*:n ollessa 0.7, *cMax*iksi on järkevää valita 1,47 – toiset alkuperäisessä julkaisussa olleet arvot ovat 0.8 ja 1.62. [Cl06a] On myös ehdotettu, että *c1* ja *c2* vaihtuisivat ajan mukaan (engl. Time Varying Acceleration Coefficients, TVAC). [Rat04]

Perusversion parannusta on yritetty mm. evoluutionäärisillä laskentamenetelmillä. Eräässä tutkimuksessa käytettiin geneettistä ohjelmointia etsimään PSO:lle päivityskaavaa, jolla saataisiin mahdollisimman hyviä tuloksia. Eri vaikutustekijät, minkä vaikutusmäärää geneettisellä ohjelmoinnilla haettiin, olivat nopeus, *pbest* ja *gbest* sekä parven massakeskipiste sekä parven hajautuneisuus eli partikkelien keskimääräinen etäisyys massakeskipisteestä. Muutaman tunnin laskennalla löydettiin PSO-versioita, jotka suoriutuivat perus-PSO:ta paremmin. [Pol05] Toisessa tutkimuksessa tarkasteltiin, kuinka usein kukin partikkeli päivitetään yhden iteraation aikana, missä järjestyksessä päivitykset

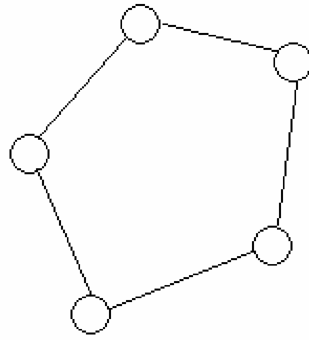
tehdään ja mikä on optimaalinen parven koko. Tutkimuksessa käytetty PSO poikkeaa perinteisestä siinä, että parven koko voi muuttua suorituksen aikana. Lisäksi partikkelin sijainti päivitetään käyttäen meneillään olevan iteraation aikana tapahtuneita muiden partikkelien päivityksiä, jotka vaikuttavat *gbest*-arvoon; ei siis rajoituta viime iteraation tuloksiin. Menetelmän tulokset saatiin koodaamalla päivitykset partikkelin numeron mukaan kromosomiksi ja etsimällä kromosomeiden mukaan suoriutuvaa parasta PSO:ta. Tuloksien mukaan partikkeleita tulisi päivittää yleisesti ottaen järjestyksellä parhaimmasta huonompaan ja parhaimpia tulisi päivittää useammin kuin huonoimpien. [Dio07]

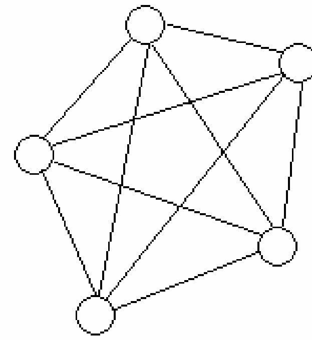
4.3 Naapurustotopologiat

Parviälyn keskeinen ominaisuus, sosiaalisuus, vaikuttaa siihen, miten informaatio kulkee partikkelien välillä. Se määrittyy naapuruston rakenteen, eli naapurustotopologian, mukaan. [Cl06a] Tässä tutkielmassa naapurustotopologialla ei siis tarkoiteta partikkelien hakuavaruuden sijainnin perusteella määräytyviä naapurustoja vaan viestintäyhteyksien perusteella muodostuvaa sosiaalista verkkoa.

Tiedon etenemistavalla ja -nopeudella on merkittävä vaikutus partikkelien liikkeisiin parvena. Jos tieto etenee hitaasti, niin sosiaalisesti toisistaan kaukana olevien partikkelien on mahdollista hajaantua ympäri hakuavaruutta, sillä kaukaisen naapuruston parhaan sijainnin vaikutus on mitätön. Näin vältytään siltä, että kaikki partikkelit käyttäytyisivät samalla tavalla, mikä ei ole suotavaa erityisesti vaikeissa ongelmissa. Toisaalta, mitä enemmän partikkelin tiedonkeruu riippuu lähinaapurustosta, sitä suurempi on riski, että tutkitaan alioptimaalista aluetta – vaikka jokin partikkeli sen jo tietäisi. Joka tapauksessa partikkelin löytäessä hyvän sijainnin tulisi tieto siitä saada levitettyä jossain vaiheessa kaikille partikkeleille. [Cl06a]

Yleisimmin käytettyjä naapuritopologioita on Kennedyn ja Eberhartin mukaan kaksi. Ensimmäisessä niistä (*lbest*) on lista, jonka päät on yhdistetty ympyräksi, jossa jokaisen partikkelin viereiset K partikkelia ovat sen naapurusto mukaan luettuna partikkeli itse. Toinen (*gbest*) on erikoistapaus ensimmäisestä; partikkelin naapureina ovat kaikki partikkelit. Kuva 3 havainnollistaa em. naapurustojen rakennetta. [Ken01]





Kuva 3. Naapurustotopologiat. Ympyrä tarkoittaa partikkelia ja viiva partikkelien välistä informaatioyhteyttä **a)** *lbest* **b)** *gbest*. [Ken01]

Eräässä Clercin ehdottamassa versiossa naapurusto puolestaan valitaan joka iteraatiolla uudelleen sattumanvaraisesti. Tälle topologialle on suoraviivaista laskea todennäköisyys, missä ajassa informaatio kulkeutuu kaikkialle, jos oletetaan naapuruston koko pysyvästi *K*:ksi. [Cl06a]

Perus-PSO:sta on kehitetty uusi versio, jossa partikkelin paikkaan vaikuttavat joka iteraatiolla kaikki sen naapurit (Fully Informed Particle Swarm, FIPS) – ei siis pelkästään se partikkeli, jolla on paras paikka. Perusajatuksena on hyödyntää mahdollisimman paljon informaatiota hyvistä paikoista. Jokaisen naapuruston jäsenen, *k*, vaikutusmäärää laskettavalle pisteelle kuvataan *W(k)*:lla; se voidaan määrittää esimerkiksi partikkelin *pbest*-sijainnin evaluoidusta arvosta ja etäisyydestä nykyiseen paikkaan. Seuraavassa *pbest* ja *gbest* -termien sijaan käytettävän p_m:n laskemiseen tarvittavat kaavat:

$$c_k = rand(0, \frac{cMax}{|N|}), \forall k \in N$$

$$p_m = \frac{\sum_{k \in N} W(k) \cdot c_k \otimes pbest(k)}{\sum_{k \in N} W(k) \cdot c_k},$$

missä *N* on naapurusto, *rand(min, max)* antaa satunnaisluvun *U*[*min*, *max*]-jakaumasta ja $\otimes$ on pistetulo. [Men04]

Testien mukaan FIPS suoriutuu useimmiten perusversiota paremmin. Testeissä kokeiltiin eri topologioiden käyttöä, ja niiden perusteella parhaiten toimivat topologiat ovat ympyrä (vertaa *lbest*, missä K=2) ja neliö (neljä joukkiota, joissa jokaisessa viisi partikkelia ja jotka on yhdistetty toisiinsa yhden partikken kautta). [Men04]

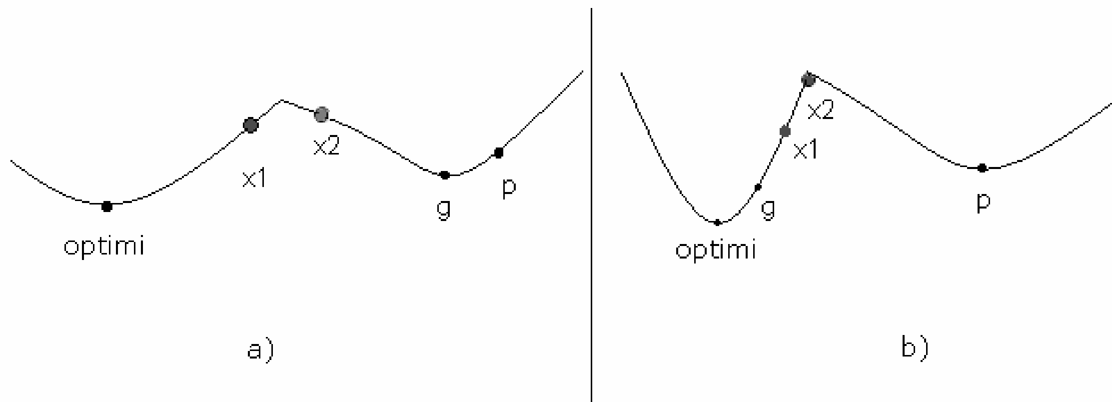
Tiedon leviämiseen voidaan vaikuttaa myös käyttämällä kahta erityyppistä partikkelia perusversion yhden sijaan. Ensimmäinen niistä on muistipartikkeli (engl. memory), joka on parhaiden paikkojen säilöjä eikä varsinaisesti niiden etsijä. Toinen tyyppi, vaeltaja (engl. explorer), puolestaan ei tallenna tietoa omista suorituksista ollenkaan, vaan tiedottaa niistä muistipartikkelia, joka puolestaan voi ohjata vaeltajia. Muisti- ja vaeltajapartikkeleilla voidaan luoda uusia entistä heterogeenisempia tiedonlevitysverkkoja, mikä mahdollistaa räätälöidyn lähestymistavan eri ongelmiin. [Cl06a]

4.4 Yliammuntaongelma

Ainakin parviälyn perusversiossa ilmenee ns. yliammuntaongelma (engl. overshooting); partikkeli pyrkii poispäin globaalista optimista liikeyhtälön vaikutusvoimien takia. Tämä ongelma ilmenee seuraavissa kahdessa tilanteessa:

1) Partikkelin oma *pbest* ja parven paras *gbest* sijaitsevat partikkeliin nähden vastakkaisessa suunnassa kuin globaali optimi (Kuva 4a).
2) Joko *pbest* tai *gbest* vaikuttaa toista voimakkaammin liikeyhtälössä ja globaali optimi sijaitsee *pbest*- ja *gbest*-sijaintien välissä. Lisäksi suuremman voiman antava vaikutustekijä on toisella puolella kuin optimi (Kuva 4b), sillä päivitysyhtälössä osatekijän vaikutuksen voimakkuus määräytyy tekijäpaikan (p tai g) etäisyydestä nykyisestä paikasta ($|x-p|$ tai $|x-g|$).

[Liu05]



Kuva 4. Yliammuntaongelma. **a)** Optimi on vastakkaisessa suunnassa vaikutusvoimiin nähden. Partikkeli siirtyy paikasta *x1* paikkaan *x2*. **b)** *p(best),* joka on toisessa suunnassa kuin optimi, on kauempana kuin *g(best)* ja vaikuttaa partikkelin nopeuteen enemmän, minkä vuoksi partikkeli pyrkii poispäin optimista, *x1*:stä *x2*:seen. [Liu05]

Ongelman ratkaisemiseksi voidaan muuttaa vaikutuskertoimia, mutta tällöin pitäisi tuntea ongelma-aluetta etukäteen. Toisaalta voidaan käyttää apuna jotain tunnettua ylimääräistä ratkaisuja generoivaa menetelmää, mutta sellaisen käyttö lisää laskentavaatimuksia mahdollisesti liikaa. [Liu05]

Ongelmaa voidaan lievittää yksinkertaisella perus-PSO:n laajennoksella MeSwarm (Memetic PSO). Se käyttää satunnaistettua, tehokasta adaptiivista askelkokoa käyttävää mäellekiipeämistekniikkaa nimeltä SW (Soliksen ja Wetsin paikallishakustrategia). MeSwarm tutkii ajoittain partikkelin lähialueen, jolloin mahdollinen paikallinen yliammuntatapaus paljastuu ja partikkelin *pbest* ja mahdollisesti *gbest* päivittyvät partikkelin uuden paikan mukaiseksi lokaaliksi optimiksi, ja ylläoleva ongelmatilanne ei ole enää voimassa. Globaalin optimin löytymistä ei voida taata siltikään, koska SW on paikallinen haku. [Liu05]

MeSwarmia käytettäessä partikkelin liikuttua normaalipäivityksen mukaisesti sovelletaan SW:tä partikkelin uuteen paikkaan todennäköisyydellä *Ps*. Menetelmä antaa testien mukaan perus-PSO:ta parempia ratkaisuja testatuille käytännön ongelmille. [Liu05] *Ps*-arvon merkitystä ei julkaisussa kerrota, mutta sen tarkoituksena lienee keventää laskentataakkaa ja mahdollisesti estää liian innokasta lokaaliin optimiin siirtymistä.

4.5 Monitavoiteongelmat

PSO:ta on kokeiltu soveltaa myös ongelmiin, joissa on useita tavoitefunktioita – mahdollisesti myös evaluointifunktioita. Tällaisissa ongelmissa haetaan ratkaisua, jolla saavutetaan kaikki tavoitteet (engl. multiobjective optimization). Ongelma on, että ratkaisujen keskinäinen vertailu voi olla mahdotonta, sillä esimerkiksi ratkaisu *A* voi olla parempi kuin *B* evaluointifunktiolla *f*, mutta *B* voikin olla parempi kuin *A* *g*:n mukaan. Ongelman osittaiseksi helpottamiseksi voidaan käyttää *Pareto*-optimaalisuuskäsitettä: sanotaan, että ratkaisu *A* hallitsee ratkaisua *B*, jos *A* on vähintään yhtä hyvä kuin *B* kaikissa tavoitteissa (evaluointifunktion mukaan) ja parempi vähintään yhdessä. Ideana on löytää hakuavaruuden kaikki ne ratkaisut, joita ei hallitse yksikään muu ratkaisu. Tätä joukkoa kutsutaan Pareto-joukoksi ja sen kuvaa evaluointifunktioissa Pareto-eturintamaksi (engl. front). Tässä tutkielmassa multitavoiteongelmiin kehitetyt PSO-menetelmät kuitenkin sivuutetaan, vaikka ne ovat tärkeitä ja niitä tutkitaan aktiivisesti. [Kod07]

4.6 Dynaamiset ongelmat

Reaalimaailman ongelmat ovat usein dynaamisia, mikä tarkoittaa, että ongelman yksityiskohdat saattavat muuttua ratkaisemisen aikana. Perus-PSO suoriutuu dynaamisista ongelmista hyvin, jos ongelman muutosnopeus on vähäistä. Kuitenkin jos muutokset ovat suuria, PSO:lla on vaikeuksia saada hyviä tuloksia. Suorituskykyä voidaan parantaa esimerkiksi alustamalla parven ominaisuuksia, kuten *pbest*-arvoa, satunnaisesti uudestaan sen jälkeen, kun huomataan, että muutoksia on tapahtunut. [Bla02]

Varautuneet partikkelit -menetelmässä (engl. Charged Particles) parvia on kaksi: toinen koostuu pelkästään neutraaleista ja toinen varautuneista partikkeleista. Varautuneisiin partikkeleihin sovelletaan klassisesta fysiikasta tunnettua lakia, joka sanoo kahden samalla tavalla varautuneen varauksen vaikuttavan toisiinsa hylkivästi voimalla, jonka suuruus on käänteinen varauksien etäisyyden neliöön. Varautuneen partikkelin *i* nopeudenpäivitys on suurinpiirtein perus-PSO:ta vastaava, mutta siihen lisätään hylkimisvoima *a*, joka saadaan laskemalla kaikkien muiden varautuneiden partikkelien *j* vaikutus *i*:hin:

$$a_i = \sum_{i \neq j,\, p_{core} < r_{ij} < p} \frac{Q_i Q_j}{{r_{ij}}^2},$$

missä Q:t ovat partikkelien varaukset ja p_{core} estää arvon liiallisen kasvun r_{ij}:n ollessa nollan lähellä; p:llä määrätään varauksien vaikutusetäisyys. [Bla02]

Menetelmän etuna satunnaisalustukseen on, ettei ajonaikana tarvitse erikseen tutkia, onko muutoksia tapahtunut. Varauksellisia partikkeleja käyttävä PSO on muutenkin hyvä ehdokas dynaamisten ongelmien optimointiin. [Bla02]

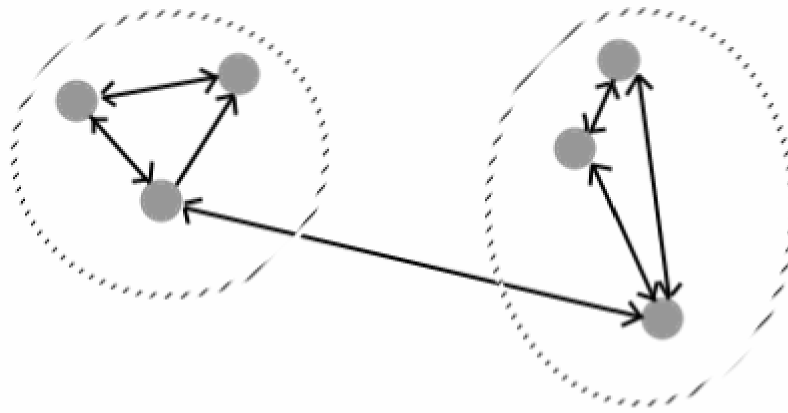
4.7 Adaptiivisuus

4.7.1 Yleistä

Käyttäjän kannalta on rasittavaa, että PSO-menetelmälle täytyy antaa parametreja (esim. parven koko), sillä käyttäjän täytyy nähdä vaivaa löytääkseen sopiva arvo eri ongelmille. Niinpä parametrit haun aikana sopiviksi muuttava – eli adaptiivinen – menetelmä on varteenotettava vaihtoehto. Toisaalta, jos ongelma-alue on etukäteen tunnettu, menetetään jonkin verran haun tehokkuutta, sillä parametrit eivät ole heti haun alussa optimaaliset. Kaikkia parametreja ei kuitenkaan voida valita ajonaikaisesti, vaan PSO:lle täytyy etukäteen kertoa ainakin evaluointifunktio sekä lopetusehto suorituksen päättymiselle. Lopetusehto voi olla suoritettava iteraatiomäärä tai arvo, joka kertoo milloin löydetty ratkaisu on tarpeeksi hyvä. [Cl06a]

4.7.2 TRIBES

TRIBES on eräs adaptiivinen PSO, jossa parvi koostuu erillisistä heimoista. Heimojen välillä on kuitenkin kahden partikkelin muodostama sosiaalinen yhteys, jotta informaatio kulkee heimojen välillä (Kuva 5). Partikkeleille määritetään laatu; partikkeli on 1) *neutraali*, jos se ei parantanut viime kierroksella *pbest*-arvoaan 2) *hyvä*, jos se paransi viime kierroksella, muttei sitä edellisellä 3) *loistava* (ja hyvä), jos paransi sekä viime että sitä edellisellä kierroksella. Partikkelin laadun mukaan määritetään heimon laatu; se on sitä parempi, mitä enemmän hyviä partikkeleita siinä on. Tarkemmin: sitä pidetään hyvänä, jos satunnainen luku nollan ja heimon koon väliltä on pienempi kuin hyvien partikkelien määrä; muutoin sitä pidetään huonona. [Cl06a]



Kuva 5. TRIBESissa partikkelit jakautuvat heimoihin. Heimon sisällä jokainen partikkeli informoi kaikkia muita heimolaisia, kun taas heimojen välillä tieto kulkee vain kahden partikkelin välityksellä. [Cl06a]

Jotta turhaa laskentatyötä ei tehtäisi, heimosta poistetaan partikkeleita, jotka ovat varmasti turhia. Niinpä hyvä heimo voi poistaa ainoastaan *pbest*-arvon mukaan huonoimman partikkelinsa. Kuitenkin, jos heimossa on vain yksi partikkeli, se voidaan poistaa vain, jos sen jollain yhteyspartikkelilla on parempi *pbest*. Yleisesti poiston yhteydessä naapurustoyhteydet päivitetään eheiksi: Jos poistettava partikkeli on yhteydessä heimon ulkopuolelle, sen yhteys siirretään heimon parhaalle. Jos sen sijaan poistettava partikkeli on ainoa heimossaan ja se on yhteydessä vähintään kahteen heimoon, esimerkiksi *A*:han ja *B*:hen, yhteydet asetetaan suoraan *A*:n ja *B*:n välille – jos niillä ei sellaista jo ole. [Cl06a]

Toisaalta, jos heimo on huono, se tarvitsee lisää partikkeleita hakuun. Tällöin jos heimolla ei ole naapureita, luodaan yksi uusi yhden partikkelin heimo. Muutoin luodaan kahden uuden partikkelin, *vapaan* ja *sidotun,* muodostama heimo. Uuden ja vanhan heimon välille luodaan naapurisuhde heimojen parhaiden partikkelien välityksellä. Uusien partikkelien sijoitus määräytyy seuraavasti. Vapaalle partikkelille valitaan sijoituspaikka hakuavaruudesta satunnaisesti kolmesta vaihtoehdosta; se sijoitetaan joko satunnaisesti hakuavaruuteen, hakuavaruuden reunoille tai hakuavaruuden nurkkiin. Ideana on, että partikkeli siirtyy haun edetessä läpi avaruuden, mikä mahdollistaa potentiaalisesti hyvien sijaintien löydön. Sidottu partikkeli puolestaan sijoitetaan vanhan heimon läheisyyteen: sen *d*-ulottuvuuden komponentti valitaan satunnaisesti $\|g_b - x_b\|$ etäisyydeltä g_b:stä, missä x_b on alkuperäisen heimon parhaan partikkelin, *x*, *pbest* ja g_b on *x*:n parhaan naapurin *pbest*. [Cl06a]

Poisto-ja lisäysmuutoksia ei kuitenkaan suoriteta joka iteraatiolla, jotta informaatio kerkeää edetä partikkelien välillä. Tehokkuussyistä valitaan yksinkertainen arvio siitä, miten nopeasti tieto leviää kaikille. Sen avulla saadaan heuristiikka, jonka mukaan seuraavaksi heimojen rakennetta muutetaan $\frac{L}{2}$ iteraation päästä, kun L on viimeisimmän rakennemuutoksen jälkeinen informaatioyhteyksien määrä. [Cl06a]

TRIBES ei käytä partikkelien siirtymisessä eksplisiittistä nopeutta, vaan paikka määräytyy todennäköisyysjakaumasta. Jakaumat ja niihin liittyvät partikkelin sijoitustrategiat ovat käytetyssä versiossa *Local by Independent Gaussian* (LIG), *Pivot* ja *Disturbed Pivot*. LIG:n ideana on etsiä läheltä parasta paikkaa käyttäen normaalijakaumaa. Partikkelin sijoituskaava on:

$$x_d \leftarrow g_d + alea_{normal}(g_d - x_d, \| g_d - x_d \|),$$

x on partikkelin sijainti, d ulottuvuus, g paras partikkelin tuntema paikka; $alea_{normal}(\mu, \sigma)$ antaa satunnaisluvun normaalijakauman $N[\mu, \sigma]$ mukaan. [Cl06a]

Pivot-sijoitusmenetelmässä puolestaan partikkelin *pbest*-sijainnin p ja naapuruston parhaan g:n avulla muodostetaan hyperpallo molempiin paikkoihin p ja g, siten että hyperpallon säde on p:n ja g:n välinen etäisyys. Menetelmässä valitaan satunnainen piste molemmista hyperpalloista ja niiden vaikutukselle annetaan painokerroin. Kerroin määräytyy suhteessa arvoon, jonka optimoitava funktio antaa ko. hyperpallon keskipisteelle. Seuraavassa on määritelty sijoituskaava, kun *alea*-funktio antaa satunnaisen paikan hyperpallosta. Lisäksi annettuna on eräät mahdolliset vaihtoehdot painokertoimille a ja b:

$$x \leftarrow a \cdot alea(H_p) + b \cdot alea(H_g)$$

$$a = \frac{f(p)}{f(p) + f(g)}, \quad b = \frac{f(g)}{f(p) + f(g)}$$

[Cl06a]

Disturbed pivot on muuten samankaltainen kuin pivot-menetelmä, mutta lopputulokseen lisätään häiriötä käyttäen normaalijakaumaa, jossa

$$\mu = 0$$

$$\sigma = \frac{f(p) - f(g)}{f(p) + f(g)}$$

Ylläolevan mukaisesta normaalijakaumasta saatu satunnaisluku *b* vaikuttaa partikkelin sijaintiin *x*, ulottuvuudessa *d*, seuraavasti:

$$x_d \leftarrow (1+b)x_d \text{ ,}$$

missä x_d on *pivot*-menetelmän mukaisesti saatu sijainti. [Cl06a]

Strategia valitaan partikkelin kolmen viimeisimmän iteraation perusteella; mitä paremmin partikkeli on suoriutunut, sitä suuremmalla alueella se saa hakea. Toisaalta ihan parhaiten käyttäytyvien partikkelien on syytä tutkia lähialue tarkasti, jotta optimin läheisyydessä päädytään optimiin. Suorituskyvyn muuttumista kahden iteraation välillä merkitään -:lla, jos partikkelin paikka huonontui 2) +:lla, jos parantui ja 3) = jos se pysyi samana. Kun otetaan huomioon partikkelin kolme viime iteraatiokertaa, saadaan yhdeksän erilaista mahdollista suorituskyvyn muutostapausta. Niiden perusteella määrätään, mitä sijoitusstrategiaa käytetään (Taulukko 2). [Cl06a] Esimerkkinä $-+$ tarkoittaa tapausta, jossa ensin partikkelin sijainti on huonontunut ja seuraavalla iteraatiolla taas parantunut; tällöin valitaan ko. rivillä oleva strategia, eli disturbed pivot.

Taulukko 2. Suorituskyvyn muutoksen vaikutus sijoitusstrategian valintaan. [Cl06a]

Suorituskyvyn muutos	**Sijoitusstrategia**
$--$ / $=-$ / $+-$ / $-=$ / $==$	*Pivot*
$+=$ / $-+$	*Disturbed pivot*
$=+$ / $++$	*Local by Independent Gaussian*

TRIBESin toimintaa voi kuvata seuraavasti: alussa yksi huonosti suoriutuva heimo luo naapuriheimon. Iteraatioiden myötä heimojen määrä kasvaa eksponentiaalisesti, kunnes partikkelit löytävät parempia arvoja. Tällöin heimot alkavat karsia kokoaan ja lopulta heimot kutistuvat yhteen yhdenhengen heimoon. Testien mukaan TRIBES toimii usein perus-PSO:ta paremmin adaptiiviisuudestaan huolimatta. [Cl06a]

4.8 Rajoiteoptimointi

PSO:ta on kokeiltu myös soveltaa rajoiteongelmiin, joissa hakuavaruudelle annetaan erilaisia rajoitteita. Ongelmat ovat muotoa [Kro04]

$$\min\ f(x),\quad g(x) \le 0\ \textit{tai}\ h(x) = 0.$$

Rinnakkaisevoluutiomenetelmää (engl. co-evolution) käyttävä Co-PSO ratkaisee rajoiteongelmia muunnettuna min-max-asetelmaan, missä duaaliongelma on saatu LaGrangen kaavalla. Menetelmässä kaksi eri parvea etsivät optimiratkaisuja liikkuen eri muuttujien suhteen; toinen parvi minimoi ja toinen maksimoi yhteistä evaluointifunktiota. Parvia liikutetaan vuorotellen ja niiden välinen vuorovaikutus saadaan aikaiseksi *gbest*- ja *pbest*-arvojen päivityksellä: kun parvi *A* on edennyt yhden iteraation, se on päivittänyt partikkelien parametreja, jotka ei muutu parvessa *B*; sen takia näitä uusia sijainteja käyttäen päivitetään *B*:n *pbest*-arvot. Vastaavasti *A*:n *pbest*-arvoja päivitetään *B*:n iteraation jälkeen. [Kro04]

4.9 PSO:n analysointi

PSO:ssa käytettävien parametrien tai ylipäänsä eri liikekaavojen analysointi on varsin haasteellista, koska partikkeleita on useita kymmeniä ja niiden käyttäytyminen on (pseudo)satunnaista. Analysointi on tarpeen, jotta voidaan ymmärtää, miten voidaan luoda uusia, parempia PSO-versioita. Halutaan myös taata, että ne suoriutuvat hyvin tietynkaltaisissa ongelmassa, sillä pelkkä eri ongelmatapausten läpikäynti ei riitä todistamaan yleistä käyttäytymistä. Todistuksia varten tarvitaan formaaleja malleja. [vdB02]

Analysointia varten joudutaan tekemään parametrien karsimista ja muuta yksinkertaistamista, jotta tutkija voi ymmärtää kokonaisuutta ja jotta se olisi ylipäänsä mahdollista laskennallisesti. Esimerkiksi on tutkittu partikkelin sijainnin todennäköisyysjakauman ominaisuuksia [Pol07] tai sitä, miten partikkeli käyttäytyy silloin, kun se ei löydä parempaa sijaintia, eli partikkeli on jäänyt tietylle alueelle (engl. stagnation) [Cl06b, Jia07]. Analysointi voidaan suorittaa esimerkiksi Bayesilaisella optimointimallilla (engl. Bayesian Optimization Model) [Mon05] tai tarkastelemalla vain kahden partikkelin muodostamaa parvea [Cl06a].

On formaalisti todistettu myös joillekin PSO-versioille, että partikkelit lähestyvät (tai eivät) jotain pistettä, eritoten optimia. Esimerkiksi perus-PSO ei välttämättä päädy lokaaliin optimiin [vdB02], mikä johtuu PSO:n käyttäytymisestä, kun $x(t) = y(t) = lbest$. Tällöinhän partikkelin nopeus riippuu ainostaan *c1*-kertoimesta – tai *w*:stä, jos käytetään iteraatiosidonnaista kerrointa. Ongelma voidaan ratkaista käyttämällä eri päivityskaavaa naapuruston parhaalle partikkelille, jolloin partikkeli hakee *lbest*-sijainnin lähiympäristössä (Guaranteed Convergence PSO, GCPSO):

$$x(t+1) = lbest + w \cdot v(t) + p(t) \cdot (1 - 2 \cdot rand),$$

missä $p(t)$:n suuruutta muutetaan sen mukaan, millaisia ratkaisuja on löytynyt; *rand* antaa satunnaisen luvun $U[0, 1]$:n mukaan. [Mes06]

GCPSO lähestyy lokaalia optimia iteraatioiden kasvaessa yhden optimin sisältävissä ongelmissa. Sen sijaan usean optimin funktioille GCPSO ei ole globaali hakualgoritmi. Perus-PSO saadaan kuitenkin lähestymään globaalia optimia esimerkiksi lisäämällä parveen satunnaisesti ajoittain paikkaansa vaihtava partikkeli (RPSO). [vdB02]

4.10 Käytännön sovelluksia

Parvioptimointia on sovellettu menestyksekkäästi useisiin eri käytännön optimointiongelmiin. Eräs niistä on hahmontunnistuksissa käytettävän neuroverkon (ks. luku 6.2) painojen sopivien arvojen löytäminen. Sitä on käytetty esimerkiksi kasvaimentunnistuksessa ja jännitteensäädössä. Vaikka jälkimmäinen ongelma sisälsi sekä jatkuvia että diskreetteja muuttujia, ratkaisu luonnistuu PSO:lta. Onnistuneiden käyttöönottojen ansiosta parvioptimointi on korvannut nopeutensa vuoksi osittain neuroverkon opetuksessa perinteisesti käytetyn virheen takaisinsyöttö -menetelmän (engl. Error Back-Propagation). [Ken01] Toinen hahmontunnistukseen liittyvä PSO-variaatio on SOSwarm-algoritmi. Se luokittelee itseorganisoituvien neuroverkkojen (engl. Self-Organizing Maps, SOM) tavoin syötteenä saadun moniulotteisen vektorin visuaaliseksi 2D-kuvaukseksi. Se ei käytä ulkopuolista opettajaa, joka kertoisi, milloin syötteestä annettu tulos on halutunlainen. [O'Ne06]

PSO:n pohjalta on luotu myös robotteja, jotka hakevat reaalimaailmassa kohteita (dPSO). Erityisen hyödyllisiä robotit ovat, kun kohde on vaikea löytää tai vaarallinen, esimerkiksi pommi. Etsintä voidaan PSO:n ansiosta hajauttaa useille pienille, edullisille roboteille. Edullisia ne ovat viestinnän ja laskennallisen tehokkuuden vuoksi, sillä robotit tarvitsevat paikkansa laskemiseksi muualta ainoastaan *gbest*-arvon eikä laskukaava ole muutenkaan laskennallisesti raskas. Testien mukaan dPSO löytää tehokkaasti kohteensa ja lisäksi skaalautuu robottien määrän suhteen ongelmitta, sillä viestintämäärä pysyy vähäisenä. [Her06]

5 Parvioptimointi ja usean optimin ongelmat

5.1 Yleistä

Multimodaaliset, eli useasta optimista muodostuvat funktiot, sisältävät graafisesti tarkasteltuna useita laaksoja. Haun aikana halutaan mahdollisesti tutkia kaikkia tai ainakin useimpia laaksoja ja niiden optimikohtia yhtä aikaa. Joskus jopa halutaan haun ratkaisuksi kaikki optimiratkaisut. Ongelmana on, että heuristiikkoja täytyy yleensä muuttaa, jotta ne pyrkisivät säilyttämään vaihtoehtoja. Tätä ongelmaluokkaa varten kehitettyjä tekniikoita kutsutaan lokeroinniksi (engl. niching). Valitettavasti lokerointia käytettäessä käyttäjän tarvitsee usein antaa myös lisäparametri lokeroiden jakoa varten. [Bir06]

5.2 Lokerointimenetelmät (NichePSO, SPSO)

Perus-PSO ei suoriudu hyvin multimodaalisessa ympäristössä, sillä partikkelit konvergoituvat (engl. converge) liian nopeasti kohti samaa pistettä, ja näin ollen haku ei ole laaja-alaista. On kuitenkin pystytty kehittämään PSO-lokerointimenetelmiä, joista osa jopa päättelee haun aikana lokerointiin tarvittavat parametrit. Eräs tapa (engl. NichePSO) on jakaa parvi useaan lokeroon siten, että eri osissa olevat partikkelit eivät pysty lähettämään tietoa omista parhaista sijainneistaan muissa osissa oleville partikkeleille. Eri lokerossa olevien parvien naapurustot ovat siis toisistaan täysin erilliset. [Bir06]

Kennedyn [Ken00] ehdottamassa menetelmässä partikkelit muodostavat rykelmiä jollakin *k-means*-algoritmilla. Artikkelissa käytetään *fastclusin*-nimisen menetelmän muunnelmaa, joka valitsee alustavasti k kappaletta tasaisesti avaruuteen levittäytynyttä partikkelia keskuksiksi; seuraavaksi algoritmi laskee kaikkien partikkelien etäisyydet keskuksiin ja asettaa jokaisen partikkelin sen lähimpään keskukseen; tämän jälkeen lasketaan jokaiselle keskukselle sen partikkelien keskipiste. Algoritmia jatketaan asettamalla lasketut keskipisteet uusiksi keskuksiksi ja toistamalla yllä kuvatulla tavalla uudestaan, kunnes keskuksien sijainti vakiintuu. [Ken00] Varsinaisessa PSO-algoritmissa keskikohtia käytetään partikkelin henkilökohtaisen parhaan paikan tai globaalin parhaan paikan sijasta. Menetelmän heikkoutena on laskennallinen raskaus, eikä laskettu keskikohta välttämättä ole paras valinta liikeyhtälön vaikutustekijäksi. [Li04]

Lokerointimenetelmissä yleisesti tarvitaan parametri, joka määrää mihin lokeroon tai rykelmään kukin partikkeli kuuluu. Parametriksi voidaan valita lokeroiden määrä tai etäisyysmitta, joka määrää, miten kaukana kaksi samaan lokeroon kuuluvaa partikkelia voivat olla. Toivottavan tuloksen saamiseksi tällaisen parametrin valitsemisen edellytyksenä on, että tunnetaan ongelma-avaruutta; ilman tietoa siitä jäädään mahdollisesti lokaaliin optimiin (liian vähän partikkeleita lokeroa kohden) tai löydetään vain osa optimeista (useat lokerot peittävät samoja optimeita). [Bir06]

SPSO:ssa (engl. The Species-Based PSO) partikkeleiden lokerointi perustuu lajien käsitteeseen. Kullakin lajilla on siemenpartikkeli, joka on lajinsa paras *pbest*-arvon mukaan ja jonka *pbest*-arvoa käytetään lajinsa sosiaalisena vaikutustekijänä (*lbest*). Kukin partikkeli kuuluu täsmälleen yhteen lajiin, mikä määräytyy euklidisen etäisyyden mukaan; partikkelit, jotka ovat r_s:n etäisyydellä siemenpartikkelista kuuluvat samaan lajiin. Partikkelit jaetaan lajeihin järjestämällä ne ensin listaan *pbest*-arvon mukaan (paras on listan kärjessä). Sen jälkeen listaa käydään läpi partikkeli kerrallaan, ja jos käsiteltävä partikkeli *A* on annetun r_s-parametrin etäisyydellä jostakin siemenpartikkelista, lisätään se tämän lajiin. Muutoin *A*:sta luodaan uusi siemenpartikkeli. Kennedyn versioon nähden SPSO vaatii vähemmän laskentatehoa, lajin siemen (*lbest*) on aina paras joukossaan ja lokeroiden määrä määräytyy automaattisesti mukautuen haun aikana. Tosin r_s täytyy määrätä – sen arvoksi valitaan yleensä kymmenes- tai kahdeskymmenesosan väliltä ulottuvuuden lukualueen suuruudesta. [Li04]

5.3 Gregarious-PSO (G-PSO)

G-PSO (seurallinen PSO, engl. Gregarious PSO) käyttää varsin poikkeavaa lähestymistapaa perusversioon nähden; partikkelit käyttävät nopeusvektorin laskemiseen ainoastaan muilta partikkeleilta saatua tietoa. Ne eivät siis hyödynnä omaa hakuhistoriaansa, lukuunottamatta omaa nykyistä sijaintiaan. Ne liikkuvat sijainnistaan kohti globaalia parasta siihen mennessä löydettyä sijaintia. Tällaisen käyttäytymisen seurauksena partikkelit juuttuvat helposti lokaaleihin optimeihin. Ongelman ratkaisemiseksi partikkelit alustetaan satunnaiseen paikkaan ja niiden nopeus asetetaan välille [-*vMax*, *vMax*], silloin kun partikkeli tulee liian lähelle parasta paikkaa. Lisäksi käytetään reaktiivista parametria γ, joka vaikuttaa nopeuteen. Parametria γ muutetaan lineaarisesti sen mukaan, parantuiko paras globaali arvo

viime iteraatiolla. Jos paranemista tapahtui, γ-muuttujan arvoa vähennetään, jonka myötä partikkelit liikkuvat hitaammin, ja täten aluetta tutkitaan tarkemmin. Muutoin γ-arvoa lisätään, jotta haku ei tuhlaisi aikaa pienellä alueella. [Pas06]

G-PSO:n idean etuna on, että aggressiivisella parhaan paikan lähialueen tutkimisella löydetään nopeasti lupaavia ratkaisuja. Kuitenkin partikkelin paikan satunnaisella alustamisella varmistutaan hakuavaruuden tarpeeksi kattavasta läpikäymisestä. Kokeiden mukaan G-PSO löytää perusversioon nähden keskimäärin paremman arvon ja lähestyy nopeammin optimia. [Pas06]

5.4 FER-PSO ja FDR-PSO

FER-PSO (engl. Fitness Euclian Distance Ratio) lähestyy multimodaalisia ongelmia jalostamalla FDR-PSO:ta (engl. Fitness Distance Ratio). FDR-menetelmässä naapuruston vaikutusvoimana on p_n, joka korvaa siis perusversion *pbest*- ja *gbest*-vaikutusvoimat. Sen d:nnen ulottuvuuden arvo saadaan siitä naapuruston parhaasta partikkelista, joka minimoi (maksimoi maksimointiongelmassa) d:nnen ulottuvuuden hyvyys–etäisyys-suhteen eli FDR:n:

$$FDR(j,i,d) = \frac{f(pbest_j) - f(x_i)}{\left| pbest_j(d) - x_i(d) \right|}$$

Toisin sanoen p_n:n valinnassa arvostetaan partikkelia, joka on lähellä d:nnessä ulottuvuudessa ja jonka *pbest*-sijainnilla on hyvä evaluoitu arvo. FDR ei kuitenkaan ole hyvä multimodaalisissa ongelmissa. Jotta FER-PSO toimisi FDR-PSO:ta paremmin, siihen on tehty lisäyksiä ja muutoksia. Ensinnäkin siinä käytetään muisti–vaeltaja-partikkelijakoa. Muistipartikkelit koostuvat partikkelien parhaista paikoista, kun taas vaeltajapartikkelit partikkelien nykyisistä paikoista. Muistipartikkelit pyrkivät pysymään alueella, missä on jo todettu lupaavia arvoja ja ne etenevät hyvyys–etäisyys-suhteen perusteella hitaasti. Vaeltajat puolestaan etsivät uusia ratkaisuja isolla alueella lähinnä aiemman sijainnin ja nopeuden perusteella. [Li07]

Toiseksi FDR-kaavaa joudutaan muuttamaan radikaalisti. Nimetään uudelleenmuokattu FDR-kaava FER:ksi ja määritetään se seuraavasti:

$$FER(j,i) = \alpha \frac{f(pbest_j) - f(pbest_i)}{\left\| pbest_j - pbest_i \right\|}$$

$$\alpha = \frac{\left\| s \right\|}{f(p_g) - f(p_w)},$$

missä α on kerroin, joka varmistaa, että hyvyyden ja etäisyyden suhde on järkevä; p_g on populaation paras *pbest* ja p_w populaation huonoin *pbest*; $\|s\|$ on arvio hakuavaruuden koosta. Se voidaan laskea jokaisen ulottuvuuden annettujen arvorajojen (*min*, *max*) euklisena etäisyytenä. Ajatellaan siis, että erotus *max-min* on hyperkuution sivun pituus ja lasketaan origosta etäisyys hyperkuution kaukaisimpaan nurkkaan. [Li07]

FER:ssä on siis useita eroja FDR-versioon nähden. Ensinnäkin partikkelin nykyisen sijainnin x_i sijaan käytetään parasta löydettyä paikkaa $pbest_i$, sillä $pbest_i$ on vähintään yhtä hyvä kuin x_i. Lisäksi sen arvo ei heittelehdi jatkuvasti, mikä takaa vakaamman FDR-arvon ja siispä partikkelin ”päättäväisemmän” etenemisen. Toiseksi ulottuvuuskohtainen vertailu on poistettu, sillä sen seurauksena saadaan joskus kelvottomia p_n-arvoja. Sen sijaan lasketaan partikkelien välinen euklidinen etäisyys. Kolmanneksi populaation yhteisen parhaan vaikutustekijän p_g sijaan käytetään jokaiselle partikkelille omaa FER:n mukaista p_n-arvoa. Siispä jokaisella populaation partikkelilla on oma hyvyys–etäisyys-suhteen mukaan paras naapuripiste. [Li07]

Kokeiden perusteella FER toimii hyvin evoluutiopohjaisiin optimointialgoritmeihin nähden multimodaalisissa ongelmissa. Sen etuna on, ettei tarvita käyttäjän antamaa erillistä lokeroparametria. [Li07]

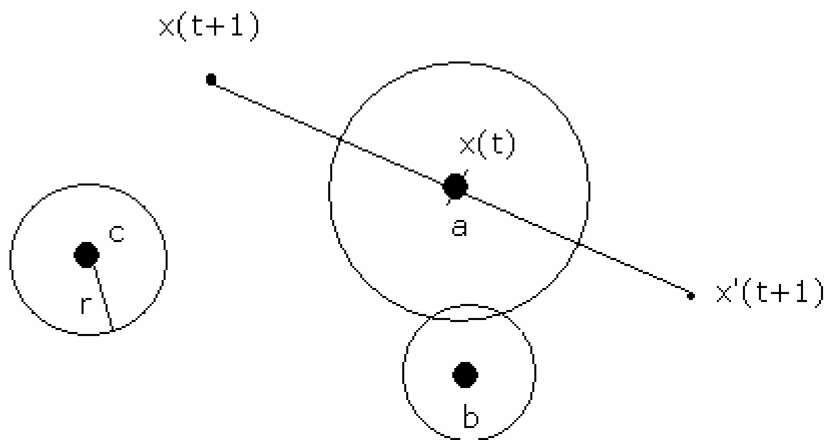
5.5 SEPSO, CRS ja CRIBS

SEPSO (engl. Spatial Extension PSO) puolestaan pyrkii vähentämään lokaaleihin minimeihin juuttumista mahdollistamalla partikkelien väliset törmäykset. Sitä varten partikkeleille määritetään säteen r määräämä pallomainen tilavuus. Kaksi partikkelia i ja j törmäävät, jos niiden ulottuvuuspinnat leikkaavat:

$$\|x_i - x_j\| \leq 2r \text{ ,}$$

missä || || operaatio on euklidinen normi. [Mon06]

Törmäyksen seurauksena partikkeli kimpoaa menosuunnasta poispäin sen verran kuin nykyisestä paikasta $x(t)$ on matkaa siihen paikkaan $x'(t+1)$, mihin partikkeli oli matkalla. Sitä havainnollistaa Kuva 6, jossa partikkeli *a* törmää *b*:hen. [Mon06]



Kuva 6. SEPSO. Partikkelin *a* ja *b* yhteentörmäys ja sen seuraukset *a*:lle. Isot täytetyt ympyrät ovat partikkelien sijainnit ja ympyrät niiden ympärillä partikkelin reunat. $x(t)$ on a:n nykyinen paikka, $x'(t+1)$ seuraava nopeuden mukainen paikka ja $x(t+1)$ törmäyksen aiheuttama varsinainen uusi paikka. Säde on kaikilla partikkeleilla sama, mutta se voi vaihdella SEPSO:n laajennoksissa CRS ja CRIBS. [Mon06]

Törmäyksen aiheuttama paikan siirtyminen ilmaistaan seuraavalla kaavalla:

$$x(t+1) = x(t) - (x'(t+1) - x(t)).$$

Lisäksi nopeus voidaan mahdollisesti kääntää:

$$v(t+1) = -v'(t+1).$$

[Mon06]

Menetelmän ideana on estää turha samankaltainen työ, mitä ilmenee, kun kaksi partikkelia etsivät pisteitä samalla alueella. Sen lisäksi tekniikkaa voidaan soveltaa lukuisiin eri PSO-

variantteihin – myös sellaisiin, joissa partikkelin paikkaa ei määritetä suoraan nopeutta muuttamalla. Huomionarvioista on kuitenkin, että säteen suuruus vaikuttaa SEPSO:n tuloksiin voimakkaasti. Erityisesti sellaisessa ongelmassa, jossa on vain yksi selkeä globaali huippu (unimodaalinen ongelma), partikkelien törmäykset vain haittaavat hyvän tuloksen löytämistä, minkä takia partikkeleille kannattaa valita pieni säde. Toisaalta ongelmassa, jossa on lukuisia optimihuippuja, säteen suuri arvo estää partikkelien juuttumisen samoihin paikkoihin. Lisäksi koko haun ajan samana pysyvä säde estää hyvien tuloksien saamisen, vaikka säde olisi määritelty lähes kuinka pieneksi tahansa ($r > 0$). Tämä johtuu siitä, että haun edetessä ja partikkelien siirtyessä yhä lähemmäksi optimia saavutetaan jossain vaiheessa $2r$-raja, minkä jälkeen tarkemman tuloksen saaminen on vaikeaa. Sen vuoksi SEPSO:n muunnelmien käyttöön lisätään jokaiselle partikkelille oma haun aikana tilanteen mukaan muuttuva säde. [Mon06]

Kun partikkelit törmäävät, niiden adaptiivista sädettä vähennetään. Tällä tavalla partikkeli pystyy pääsemään pois lokaalista minimistä, mutta samalla parannetaan mahdollisuutta tutkia aluetta yhä tarkemmin. Jokaisella partikkelilla p on törmäyslaskuri b_p ja törmäys tapahtuu, jos

$$\left\| x_i - x_j \right\| \leq (\gamma^{b_i} + \gamma^{b_j})r ,$$

missä $\gamma, \in [0,1]$ törmäyksen adaptiivisuusvakio. [Mon06]

Tämän mukaista algoritmia kutsutaan CRS:ksi (Contracting Radius SEPSO). Kokeiden mukaan ylläolevan mukaan parannettu PSO suoriutuu hyvin, mutta saattaa keskittyä liian aikaisin yhteen alueeseen. [Mon06]

Lisälaajennus CRIBS (engl. Contracting Radius, Increasing Bounce) muuttaa partikkelin säteen lisäksi sitä, kuinka kauas partikkeli sinkoaa taaksepäin törmätessään. Partikkelin sijainnin määräävä kaava on nyt seuraavanlainen:

$$x(t+1) = x(t) - \gamma^{-b}(x'(t+1) - x(t)).$$

Idea on, että kun törmäyssäde pienenee, partikkelit kasaantuvat ajan myötä yhä enemmän, ja niiden törmätessä minimistä poispääsy vaikeutuu. Sen vuoksi partikkelien

kimpoamismatka kasvaa törmäyksien lisääntyessä ja törmäys aiheuttaa partikkelin siirtymisen kauas optimista, mikä on erityisen hyödyllistä multimodaalisten funktioiden optimoinnissa; niissä CRIBS suoriutuukin hyvin kokeellisten tuloksien mukaan. Toisaalta unimodaalisissa funktioissa kasvava kimpoamismatka ja ylipäänsä törmäykset lisäävät hyödyntöntä optimista poikkeamista. Niinpä CRS ja CRIBS suoriutuvat niissä perus-PSO:takin huonommin. [Mon06]

5.6 Attractive-Repulsive PSO (ARPSO)

ARPSO (engl. Attractive-Repulsive PSO) käyttää hieman samankaltaista ideaa kuin SEPSO. Partikkeleille lasketaan hajautuneisuus parven massakeskipisteestä, ja jos partikkelit ovat liian lähellä toisiaan (hajautuneisuus < η^-), ne asetetaan hylkimistilaan; tällöin partikkelit pyrkivät poispäin toisistaan, ja täten välttävät mahdollisia paikallisia optimeita. Kun hajautuneisuus on taas tarpeeksi suuri (> η^+), parvi asetetaan takaisin normaaliin tilaan, jossa partikkelit pyrkivät toistensa luokse. Parven *S* hajautuneisuuden kaava on seuraavanlainen:

$$diversity(S) = \sum_{i=1}^{|S|} \frac{\|x_i - x_c\|}{|S|L}$$

$$x_c = \frac{\sum_{i=1}^{|S|} x_i}{|S|}$$

missä x_c keskipiste ja $|S|$ joukon koko; *L* on pisin mahdollinen diagonaali hakualueella. [Mon06] Sen voi laskea etsimällä parametrit *y1* ja *y2*, jotka maksimoivat kaavan $\|(y1 - y2)\|$, siten että *y1*:n ja *y2*:n kautta muodostuva jana on hakuavaruuden sallitulla alueella. Tässä *y1*, *y2* ovat hakuavaruuden vektoreita ja $\| \ \|$ operaatio on euklidinen normi.

Partikkelin tila määrää siis, vaikuttavatko naapurin voimat vetovoimaisesti vai hylkivästi. Tämä saadaan aikaiseksi käyttämällä sosiaaliseen vaikutusvoimaan kerrointa *s*, joka on -1 hylkimistilassa ja 1 muutoin. [Mon06]

Kokeiden perusteella ARPSO ei toimi multimodaalisissa ongelmissa niin hyvin kuin CRIBS. On hieman intuition vastaista, miksi lokaaliin päätökseen perustuva SEPSO-pohjainen ratkaisu toimii globaalia hajautuneisuustietoa käyttävää versiota paremmin.

Selitys lienee, ettei euklidisen etäisyyden käyttö sovi yleisesti hajautuneisuuden mitaksi, ja CRIBSin paikallisessa päätöksenteossa sen heikkous ei ilmene niin vahvasti. [Mon06]

5.7 Breeding Swarms

On myös kokeiltu yhdistää evoluutionäärisiä optimointimenetelmiä PSO:hon. Eräässä niistä (engl. Breeding Swarms, BS) ideana on soveltaa nopeus- ja paikanpäivitysrutiineihin valinta-, mutatointi- ja risteytysevoluutiotekniikkoja. BS-menetelmässä $\psi \cdot N$ kappaletta *pbest*-arvon mukaan huonointa partikkelia poistetaan *N*-kokoisesta populaatiosta jokaisen iteraation jälkeen. Muut partikkelit (merkitään niitä *A*:lla) säilyvät populaatiossa, ja niiden nopeus- ja paikkapäivitykset tehdään normaalisti. Poistettujen partikkeleiden tilalle luodaan uusia käyttäen *A*:sta turnausvalinnalla valittuja vanhempia, joista luodaan jälkeläisiä käyttämällä risteytysoperaatiota VPAC (engl. Velocity Propelled Averaged Crossover). Sitä soveltamalla vanhempiin *p1* ja *p2* saadaan kaksi jälkeläistä *c1* ja *c2*:

$$c_1(x_i) = \frac{p_1(x_i) + p_2(x_i)}{2} - \varphi_1 p_1(v_i)$$

$$c_2(x_i) = \frac{p_1(x_i) + p_2(x_i)}{2} - \varphi_2 p_2(v_i),$$

missä suluissa on ominaisuus (paikka *x* ja nopeus *v*) ja suljetta edeltää ominaisuuden kohde (*p1* ensimmäinen vanhempi, *p2* toinen vanhempi); *i* on vektorin komponentti (ulottuvuus) ja φ on satunnaisluku *U*[0, 1]:n mukaan. [Set05]

Lisäksi jälkeläisen jokaisella ulottuvuudella on mahdollisuus mutatoitua normaalijakauman *N*[0, *var*] mukaan todennäköisyydellä 1/*ulottuvuuksia*, missä *var*-parametria vähennetään joka sukupolven luonnin jälkeen `1 -> 0.1`. Lapsipartikkelien nopeudet puolestaan määräytyvät vanhemmilta (c_1:sen p_1:seltä ja c_2:sen p_2:selta) ja *pbest* asetetaan lapsipartikkelin nykyiseksi sijainniksi. [Set05]

Em. menetelmä luo siis kaksi lasta, jotka ovat sijainniltaan vanhempiensa välissä; lapsien nopeuden itseisarvo on vanhempiensa mukainen, mutta suunta on poispäin vanhempiensa nopeuden suunnasta. Näin saadaan osa partikkeleista leviämäään erilleen toisistaan. Tämä toimiikin kokeiden mukaan hyvin: partikkelit eivät juuttuneet yhden pisteen ympäristöön ennenaikaisesti, mutta silti haku pystyi löytämään hyvän ratkaisun nopeasti. [Set05] Myös

muita evoluutiotekniikoita PSO:hon yhdistäviä malleja on kokeiltu: geneettistä algoritmia (engl. Hybrid Particle Swarm Optimiser with Breeding and Subpopulations) [Løv01], differentiaalista evoluutio-operaatiota (engl. Hybrid Particle Swarm with Differential Evolution Operator) [Zha03] ja evoluutiostrategiaa (engl. Particle Swarm Guided Evolution Strategy) [Hsi07].

5.8 Predator-Prey Optimization

Saalistaja–saalis-optimointimallissa (engl. Predator-Prey Optimisation) partikkelien lokaalista minimistä irtipääsy saadaan aikaiseksi lisäämällä saalispartikkeleja jahtaava saalistajapartikkeli. Saalistaja pyrkii kohti parhaan *pbest*-sijainnin omaavaa partikkelia A; saaliit (kaikki pl. saalistaja) puolestaan kaikkovat saalistajaa. Koska partikkelit yleisesti pyrkivät *gbest*-sijaintia kohti (erityisesti sen löytänyttä A:ta), seuraamalla A:ta saalistaja tehokkaasti ahdistelee koko parvea. [Sil02]

Saaliin halu mennä poispäin saalistajasta riippuu pelkotodennäköisyydestä P_f, joka on ulottuvuuskohtainen arvo: Jos saalis päättää ulottuvuudessa j olla lähtemättä pakoon, se käyttää tavanomaista PSO:n päivityssääntöä, jossa nopeustekijän kerroin on lineaarisesti vähenevä w. Jos sen sijaan saalis päättää karata (ulottuvuudessa j), niin nopeuden päivityksen summaan lisätään uusi termi

$$c4 \cdot D(d), \; missä \; D(d) = ae^{-bx},$$

missä $c4$ on satunnaisluku U[0, u]:n mukaisesta mallista; u:lla voidaan vaikuttaa siihen, kuinka nopeasti saalistaja voi saada kiinni parhaan yksilön. a-parametrilla puolestaan määritetään, kuinka paljon saalistajalla ylipäänsä on vaikutusta ja b:llä, miltä etäisyydeltä saalistaja alkaa vaikuttaa merkittävästi. Kaavoista seuraa, että mitä lähempänä saalistaja on saaliista euklidisen etäisyyden mukaan, sitä voimakkaammin (eksponentiaalisesti) saalis pyrkii poispäin saalistajasta. Tällaisella määrittelyllä haun alkuaikana saalistaja ei vaikuta kuin harvoihin partikkeleihin, mutta parven löytäessä lokaalin optimin myös saalistaja poikkeaa sinne ja ajaa parven sieltä pois. Testien mukaan saalista–saalis-malli toimii huomattavasti paremmin kuin perus-PSO: parvi siirtyy kokonaisuudessaan hyvään paikkaan nopeammin ja löytää lopulta parempia arvoja. [Sil02]

5.9 Comprehensive Learning PSO (CLPSO)

CLPSO-menetelmässä (Comprehensive Learning PSO) *pbest*- ja *gbest*-vaikutusvoimien sijaan pyritään hyödyntämään kaikkia *pbest*-vaikutusvoimia. Nopeuden päivityskaava partikkelille *i* ulottuvuudessa *d* on nyt:

$$v_i(d) \leftarrow w \cdot v_i(d) + c \cdot rand \cdot (pbest_{f_i d}(d) - x_i(d)),$$

missä *w* on lineaarisesti vähenevä kerroin ja *rand* satunnainen luku *U*[0, 1]:n mukaisesti; $f_i(d)$ määrää, minkä partikkelin *pbest*-sijaintia käytetään partikkelin *i* vaikutustekijänä ulottuvuudessa *d*. Todennäköisyydellä Pc_i valittu *pbest* on partikkelin *i* oma *pbest*. Muutoin se saadaan ottamalla satunnaisesti kaksi partikkelia (muuta kuin *i*) ja valitsemalla niistä evaluointifunktion mukaan parempi. Jos käy niin, että kaikissa ulottuvuuksissa valitaan oma *pbest*, valitaan satunnaisesta ulottuvuudesta jonkun muun partikkelin *pbest*. [Lia06]

Samaa *f*-arvoa käytetään niin kauan, kunnes *m*:ään vuoroon valituilla ulottuvuus–*pbest*-kombinaatioilla ei olla saatu parannettua partikkelin sijaintia. Tällainen menetelmä parantaa haun levinneisyyttä ja suoriutuu vaikeista multimodaalisista ongelmista useita PSO-versioita paremmin. Etuna sillä on lähes yhtä yksinkertainen toimintatapa kuin perus-PSO:lla. Toisaalta Pc_i:n ja *m*:n arvojen valinta riippuu ongelmasta ja vaatii siten erityistä huomiota. [Lia06]

5.10 HPSO-TVAC

HPSO-menetelmä (Self-Organizing Hierarchical PSO), tai tarkemmin iteraatiosidonnaisia vaikutuskertoimia käyttävä HPSO-TVAC, ei käytä päivitysyhtälössä vanhaa nopeutta, vaan partikkelin nopeus alustetaan joka iteraatiolla uudestaan. Tämä mahdollistaa, että partikkeleilla on tarpeeksi liikemomenttia myös haun loppuvaiheessa, jotta päästään pois lokaalista minimistä. Partikkelin *i* nopeus ulottuvuudessa *d* päivittyy HPSO-TVAC:ssa ohjelma 4:n askelien mukaan: [Rat04]

Ohjelma 4.

```
1. vi(d) <- c1 · (pbesti(d) - xi(d)) + c2 · (gbest(d) - xi(d))
2. if vi(d) = 0){
3.   if(rand < 0.5) vi(d) <- rand · vinit
4. } else vi(d) <- -rand · vinit
5. vi(d) <- sign(vi(d)) · min(|vi(d)|, vMax)
```

Yllä olevassa ohjelmassa v_{init} on alustusnopeus. Testien mukaan HPSO on hyvin toimiva PSO-menetelmä. [Rat04]

5.11 Gaussian-Dynamic Particle Swarm (GDPS)

On esitetty myös eksplisiittisestä edellisen kierroksen nopeudesta riippumattomia PSO-versioita, joista tässä käsitellään GDPS (Gaussian-Dynamic Particle Swarm). GDPS-menetelmässä partikkelin uusi sijainti määräytyy todennäköisyysjakauman mukaan nykyisestä sijainnista. Se on siis dynaaminen: partikkelin sijainnin muuttuminen ajan suhteen nähdään liikkeenä, ja kahden peräkkäisen sijainnin perusteella saadaan nopeus. Kennedyn aiemmin esittelemässä ei-dynaamisessa todennäköisyysmallia käyttäneessä versiossa uusi sijainti määritetään sen sijaan *pbest*-sijainnin ympäristöstä. GDPS:ssä dynaaminen paikkakaava määrää uuden sijainnin nykyisestä paikasta, nopeudesta $x(t+1)$-$x(t)$ ja kaikkien partikkelien *pbest*-termien vaikutuksesta. Satunnaisgeneraattorin antamaa arvoa sovelletaan ainoastaan jälkimmäiseen ja täten voidaan laskea, kuinka todennäköisesti kullekin x:n läheisyydessä olevalle välille uusi paikka määräytyy. [Ken05]

GDPS:ssä partikkelin halutaan jäävän lähelle vanhaa sijaintiaan. Se saavutetaan, kun uuden sijainnin etäisyys vanhaan määräytyy satunnaisluvun poikkeamasta odotusarvoon ja satunnaismalliksi otetaan Gaussin normaalijakauma. Normaalijakauma soveltuu hyvin tähän, sillä se on painottunut niin, että vain pienellä todennäköisyydellä saadaan selvästi poikkeava arvo. Partikkelin paikan päivityssäännöksi saadaan:

$$
\begin{aligned}
x(t+1) = x(t)& \\
&+ W1 \cdot (x(t) - x(t-1)) \\
&+ W2 \cdot (avgp - x(t)) + G(0,1) \cdot \frac{frange}{2}
\end{aligned}
$$

$$avgp = \sum_{k=1}^{K} \frac{pbest_k}{K}$$

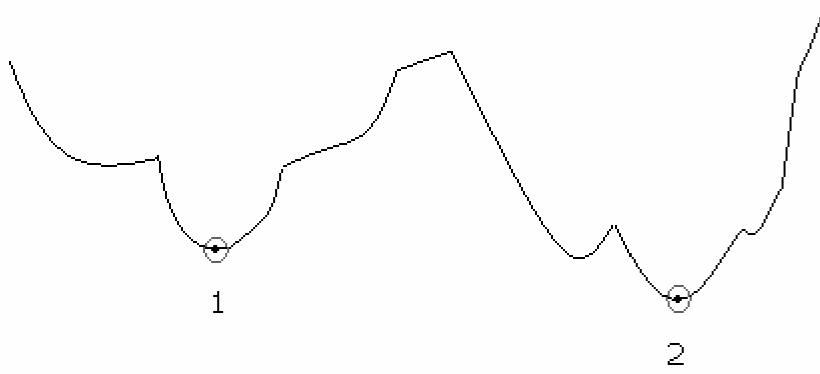
$$frange = \sum_{i=1}^{K} \sum_{k=1, i \neq k}^{K} \frac{\left| pbest_i - pbest_k \right|}{\binom{K}{2}}$$

missä *W1*, *W2* ovat vaikutustekijöiden säädettäviä kertoimia (Kennedyn testeissä käytettiin arvoja *W1* = 0,729 *W2* = 2,187, mutta ei selvitetty, miten ne oli valittu); *G*(0, 1) antaa satunnaisen luvun *N*[0, 1]-jakaumasta, *avgp* on *K*-kokoisen naapuruston *pbest*-sijaintien vaikutus ja *frange* on kerroin, joka on Kennedyn kokeiden mukaan todettu hyväksi. [Ken05]

Se, että partikkeli pystyy teoriassa siirtymään nykyisestä paikasta mihin tahansa paikkaan (normaalijakauma jatkuu äärettömyyteen), näyttää toimivan testien mukaan hyvin: GDPS pystyy välttämään ennenaikaista paikoilleen juuttumista sekä vaikuttaisi suoriutuvan yleisesti paremmin kuin mm. FIPS. [Ken05]

5.12 Monisuppiloiset ongelmat

Multimodaalisuutta vaikeampi ongelma on nk. monisuppilotyyppiset (engl. multifunnel) ongelmat. Monisuppilolle ei ole yksikäsitteistä määritystä; voidaan kuitenkin ajatella, että yksi suppilo muodostuu mahdollisesti useista lokaaleista optimeista, mutta kuitenkin niin, että keskimäärin kauempana suppilon keskuksesta olevat pisteet saavat huonompia evaluointiarvoja (Kuva 7). Monisuppiloisten ongelmien globaalin optimin löytäminen on perus-PSO:lle vaikeaa; se on selvää jo multimodaalisuuden vuoksi, sillä yksisuppiloinenkin ongelma voi olla multimodaalinen. Sen lisäksi lienee syytä selventää PSO:n vaikeuksista monisuppilo-ongelmissa hieman: Ensinnäkin parvi luultavasti jää lopuksi sille suppilolle, mihin suurin osa parvesta alussa joutuu. Toiseksi partikkelit juuttuvat tiettyyn alueeseen ylipäänsä suhteellisen helposti. Syy näihin on, että haku on yhteistyöpainotteista; partikkeli liikkuu siellä päin, missä muutkin ovat ja missä ollaan jo oltu. Nämä toiminnallisuushäiriöt PSO:ssa ovat erityisen ikäviä, sillä reaalimaailman ongelmat ovat usein monisuppiloisia. [Sut06]



Kuva 7. Kaksisuppiloinen funktio.

6 Ongelmanratkaisu parviälyllä

6.1 Tehtävänasettelu

Tässä tutkimuksessa selvitetään, kuinka hyvin parvioptimoinnilla ja rinnakkaisevoluutiolla voidaan kehittää tekoälyä Connect-lautapeliin. Tekoälyä kehitetään optimoimalla neuroverkkoa, jota käytetään staattisessa evaluointifunktiossa. Tekoälyn hyvyyttä arvioidaan pelaamalla kehitettyä tekoälyä soveltavaa pelaajaa

a) satunnaisen siirron tekevää vastustajaa
b) yksinkertaista Naive 4 -heuristiikkaa tai sen lautakoon mukaista yleistystä [Sch02] ja
c) minimax-algoritmia käyttävää pelaajaa vastaan.

Erityisesti halutaan selvittää eri parametrien vaikutusta oppimiseen, joista tärkeimmät ovat parven koko, rinnakkaisevoluution vastustajien määrä ja PSO-iteraatioiden määrä. Seuraavaksi esitellään tarvittava taustateoria tehtävänasettelun tuloksien ymmärtämistä varten.

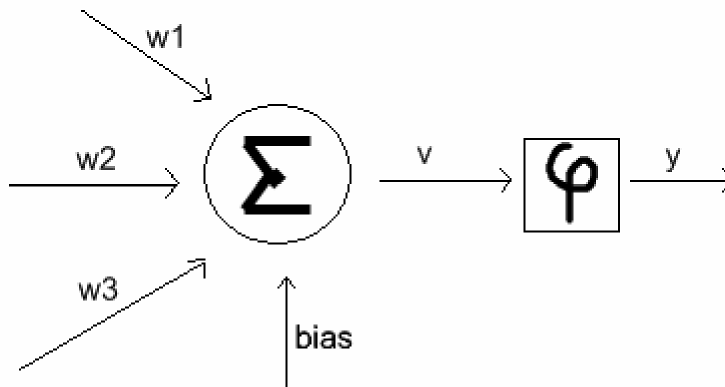
6.2 Neuroverkko

Neuroverkko (engl. Artificial Neural Network) on ihmisaivojen toimintaa mallintamalla luotu laskentamalli, joka poikkeaa huomattavasti nykytietokoneen von Neumann -arkkitehtuurista. Neuroverkko koostuu pienistä laskentayksiköistä, neuroneista, jotka on kytketty toisiinsa. Kytköksillä on paino w, joka kuvaa kytköksen kautta saatavan syötteen vaikutusta neuroniin. Neuroniin k sisääntulevien kytkösten yhteisvaikutus v_k saadaan summaamalla painotetut arvot yhteen. Neuronista uloslähtevä arvo y_k saadaan puolestaan soveltamalla summaan aktivaatiofunktiota φ. Lisäksi neuronilla on *bias* 'painotus', jolla saadaan säädettyä vaikutusta lineaarisesti (Kuva 8). *Bias* voidaan kuitenkin ajatella ylimääräisenä neuronin sisääntulona, jonka paino on yksi. Näin *biaksen* vaikutus saadaan sisällytettyä samaan summausfunktioon:

$$v_k = b_k + \sum_{j=1}^{m} w_{kj} x_j = \sum_{j=0}^{m} w_{kj} x_j$$

$$y_k = \varphi(v_k)\ ,$$

missä m on neuroniin tulevien kytköksien määrä, b_k on *bias* ja $w_{k0} = b_k$. [Hay99]



Kuva 8. Neuronin laskentamalli. [Hay99]

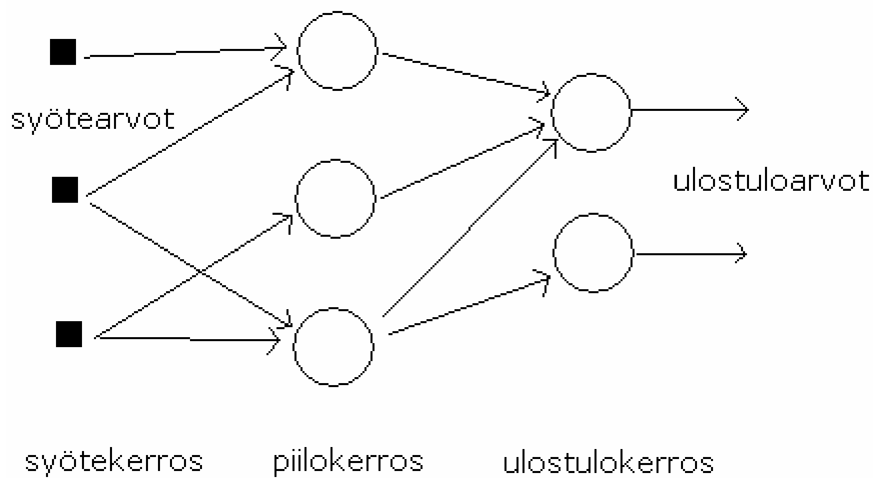
Usein käytettyjä aktivaatiofunktioita ovat kynnysfunktio (engl. threshold) ja epälineaarinen sigmoid. Kynnysfunktio on triviaali: se antaa arvon 1, jos $v_k >= 0$, muutoin 0. Eräs yleinen, tämänkin tutkielman testeissä käytettävä sigmoid-funktio on puolestaan määritelty seuraavasti:

$$\varphi(\mathrm{v}) = \frac{1}{1 + \mathrm{e}^{-\mathrm{av}}}\ ,$$

missä a määrittää S-kirjaimen muotoisen sigmoidin kaltevuuden. [Hay99]

Neuronit sijaitsevat neuroverkossa kerroksissa (engl. layer). Yksinkertaisimmillaan neuroverkossa on vain syöttö- ja ulostulokerrokset, joiden välillä on kytköksiä. Usein kuitenkin tarvitaan enemmän kerroksia (engl. multilayer) monipuolisempien kytköksien mahdollistamiseksi. Tällöin neuroneita lisätään välikerrokseen, jota kutsutaan piilokerrokseksi (engl. hidden layer). Jo yksi piilokerros kasvattaa neuroverkon laskentapotentiaalia, mutta useampia voidaan käyttää halutessa. Lisäksi neuroverkkoja on eri tyyppisiä sen mukaan, miten kytkökset kulkevat kerrosten välillä. Eteenpäinsyöttöneuroverkossa (engl. feedforward) kytkökset menevät kerrokselta seuraavaan – ei ikinä taaksepäin tai saman kerroksen neuroniin. Rekursiivisissa (engl.

recurrent) neuroverkoissa saadaan aikasidonnaista laskentaa siten, että neuroniin tulee syötteenä aiemman tai saman kerroksen neuronin, mahdollisesti neuronin itsensä ulostulo. [Hay99] Kuva 9 esittää kolmekerroksisen eteenpäinsyöttöneuroverkon rakenteen, jollaista myös tässä tutkielmassa käytetään, sillä se on mahdollisimman yksinkertainen rakenteeltaan. Lisäksi tässä tutkielmassa piilokerroksessa olevia neuroneita kutsutaan piiloneuroneiksi.



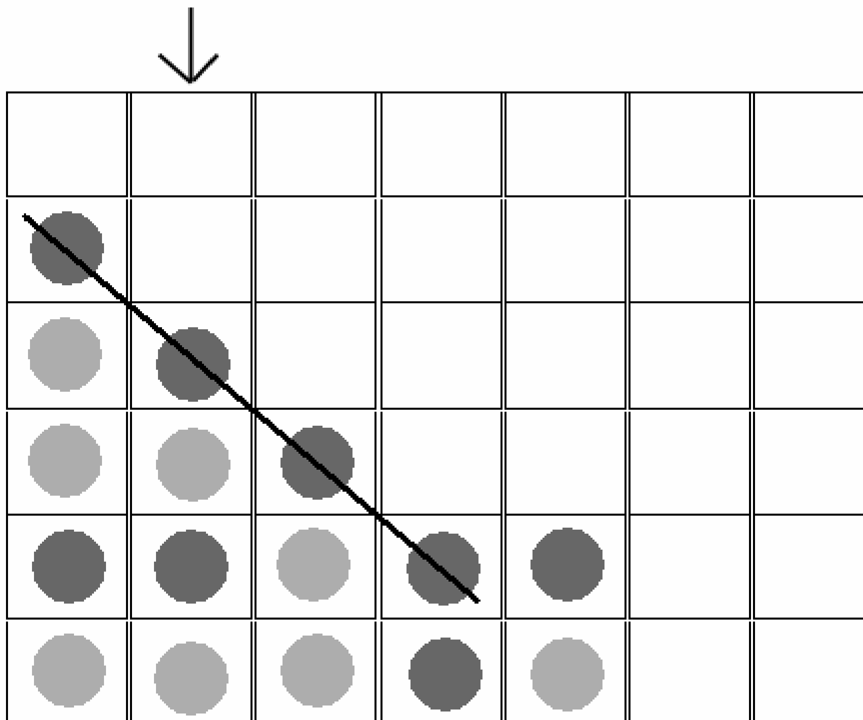
Kuva 9. Eteenpäinsyöttöneuroverkko, jossa on yksi piilokerros. [Hay99]

Neuroverkkojen vahvuus on tiedon yleistäminen oppimisen avulla. Neuroverkkoa opetetaan antamaan tietyillä syötteillä halutunlainen ulostulo, mikä tarkoittaa käytännössä, että neuroverkon painoja muutetaan niin, että aktivaatiofunktio antaa sopivan arvon. Opetusta ohjastetaan valmiiksi kerätyllä aineistolla. Yleistäminen tarkoittaa sitä, että neuroverkko kykenee antamaan haluttuja ulostuloja myös syötteille, joita ei rajallisessa opetusaineistossa ollut. Se siis löytää yksityiskohtien seasta hahmoja (engl. pattern). [Hay99] Tässä tutkielmassa käytetään yleisestä poikkeavaa menetelmää painojen muuntamiseen; sopivia painoja haetaan PSO:lla siten, että neuroverkkoja verrataan suhteellisesti toisiinsa.

6.3 Connect-4-lautapeli

Connect-4 on lautapeli, jonka standardiversion pelialue on leveydeltään seitsemän (7) ja korkeudeltaan kuusi (6). Pelaajat vuorottelevat tiputtamalla nappulan ylhäältä jostakin kokonaan täyttämättömästä sarakkeesta. Tarkoituksena on muodostaa neljän (4) peräkkäisen oman kiven suora diagonaalisesti, horisontaalisti- tai vertikaalisti (Kuva 10).

Pelille on todistettu, että aloittava pelaaja voittaa pelaamalla optimaalisesti [All88]. Pelin voittamiseksi pelaajan täytyy saada aikaiseksi joko kahdelta puolelta avoin suora tai pakotettu avoin suora. *Kahdelta puolelta avoin suora* tarkoittaa kolmen peräkkäisen oman nappulan suoraa, siten että sen molempiin päihin voidaan lisätä uusi kivi seuraavalla vuorolla. Tällöin vastustaja voi seuraavalla vuorolla estää neljän suoran muodostamisen vain toiselta puolelta. *Pakotettu avoin suora* tarkoittaa tilannetta, jossa pelaaja *A* estää *B*:n jonkin kolmen suoran laajentamisen neljäksi, mutta samalla mahdollistaa jonkin muun kolmen suoran täydentämisen *B*:n seuraavalla vuorolla. [Sch02]



Kuva 10. Connect-4-lautapeli. Pelaaja tiputti nappulan toisesta sarakkeesta. Siitä seurasi, että samanvärisiä nappuloita on neljä peräkkäin, mikä on voiton edellytys. [Sch02]

Connect-4:iin on tutkittu erilaisia tekoälyohjelmia, jotka eivät hae valmiiksi laskettua voittosiirtoa tietokannasta eri lauta-asemille. Jos aikaa miettiä on rajoitettu määrä, pelaajan täytyy tyytyä silloin alioptimaaliseen päätökseen. Eräässä tutkimuksessa ehdotetaan ratkaisumalliksi maksimaalisen odotetun hyödyn strategiaa (engl. Maximum Expected Utility [Nor03]), joka perinteisestä minmax-algoritmista poiketen pyrkii hyödyntämään vastustajan virheitä ottamalla riskejä. [Sen97]

On myös tutkittu tässä tutkielmassa käytettävää ajatusta, kuinka neuroverkko toimii staattisena evaluointifunktiona Connect-4:ssa. Testejä on tehty evoluutiopohjaisella

menetelmällä, jossa neuroverkot on koodattu geeneiksi. On kokeiltu myös kulttuurioppimisen soveltamista: uuden generaation luonnin jälkeen populaation parhaimmat yksilöt (neuroverkot) toimivat opettajina muille neuroverkoille virheen taaksepäin levitys -menetelmällä. [Cur04] Taaksepäin menetelmää käytettiin myös toisessa tutkimuksessa kehitetyssä Neural Connect 4 -ohjelmassa, mutta siinä opetuksen ohjaus pohjautuu voitettujen pelien tietokantaan. Jo kymmenellä sopivasti valitulla pelillä saatiin paras neuroverkko opetettua vahvemmaksi kuin mitä verrokkina käytetyt heuristiikkapohjaiset Connect-4-pelaajat. Lisäksi huomattiin, että on järkevämpää opettaa pienellä määrällä tarkkaan valittuja pelejä kuin valtavalla satunnaisella pelikokoelmalla. Näin vältytään väärän ja turhan tiedon opettamiselta. [Sch02]

6.4 Neuroverkon opettaminen rinnakkaisevoluutiolla käyttäen PSO:ta

Rinnakkaisevoluutio (engl. co-evolution) on yhteistyöpohjainen evoluutiomenetelmä, jossa populaation jäsenet kehittyvät toisistaan vaikutteita ottaen; evaluointifunktio määrää yksilölle hyvyyden, joka on suhteellinen arvo yksilön menestyksestä populaation verrattuna. Tämä tutkielma ottaa mallia aiemmin esitellystä menetelmästä, jossa neuroverkkojen painojen arvot määräytyvät rinnakkaisevoluutiolla ja optimaalista painoarvotuksia etsitään PSO:lla. Tutkielman menetelmässä on populaatio agentteja (partikkeleja), joilla on kullakin painojen mukaan parametrisoitu neuroverkko. Neuroverkkoa käytetään staattisena evaluointifunktiona antamalla sille syötteenä laudan nappula-asetelma ja pelaajan vuoro. Normaaliagenttien lisäksi populaatiossa on agenttien *pbest*-sijainneista muodostetut partikkelit, joiden paikkaa ei muuteta päivityssäännöillä, vaan niitä käytetään ainoastaan verrokkipelaajina. [Mes04]

Jokainen agentti pelaa toisia agentteja vastaan muutaman pelin yhden iteraation aikana. Voitosta annetaan positiivisia pisteitä, häviöstä negatiivisia ja tasapelistä 0. Ideana on, että neuroverkon painot ovat parvioptimoinnin hakuavaruus ja peleistä saatu yhteenlaskettu pistemäärä on niiden evaluoitu arvo. Painoja muutetaan PSO:n päivityssääntöjen mukaan, minkä jälkeen suoritetaan uusi pelikierros. Tätä jatketaan haluttu määrä kertoja, kunnes valitaan parhaiten menestynyt agentti. Sitä peluutetaan satunnaisesti pelaavaa, tai muuta PSO-versiolla kehitettyä agenttia vastaan, lukuisia kertoja, josta päätellään tilastollisesti, miten hyviä agentit ovat suhteessa toisiinsa. Ohjelma 5 kuvaa menetelmän rakenteen.

Ohjelma 5.

```
1. Alusta PSO:n parvi satunnaisesti
2. while(iteraatioita jäljellä)
  a) alusta agenttien suorituspisteet
  b) luo kilpailijat partikkeleista ja niiden pbest-sijainneista
  c) for (x in kilpailijat)
       i. vastustajat = k satunnaista vastustajaa x:lle
      ii. for(y in vastustajat)
            x (1. pelaaja) pelaa y:tä (2.) vastaan
            y (1. pelaaja) pelaa x:ää (2.) vastaan
            voittajan pisteitä lisätään kahdella;
            häviäjän vähennetään yhdellä;
            tasapeli ei aiheuta muutoksia
  d) laske naapuruston paras agentti
  e) for(p in parvi)
       i. päivitä p:n pbest
      ii. sovella p:n liikepäivityskaavaa
[Mes06]
```

Ohjelmalistausta mukautetaan useassa tämän työn testissä siten, että 1. ja 2. pelaajat kilpailevat omissa parvissaan. Tällöin c-kohta muuttuu siten, että valitaan 1. pelaajien parvesta jokaiselle satunnaiset vastustajat 2. pelaajien parvesta ilman, että valittuja pelattaisiin sekä 1. että 2. pelaajana. Lisäksi 2. pelaajien parvi käydään samalla tavalla läpi hakien satunnaiset vastustajat 1. pelaajien parvesta.

6.5 Testimenetelmä

6.5.1 Yleistä

Tässä työssä testataan edellä esitetyn PSO:n ja rinnakkaisevoluution yhdistelmämenetelmän toimivuutta Connect-4-pelissä. Testausta varten ensin määritetään verrokkitekoälyt. Sen jälkeen annetaan yleiskuvaus testauksesta ja siihen liittyvistä arvoista, kuten PSO-tekniikoiden parametreista. Lopuksi kerrotaan, mitä kullakin testillä on tarkoitus saavuttaa.

6.5.2 Verrokkitekoälyt

Seuraavassa on selvitys tekoälyä käyttävistä pelaajista, joiden keskinäistä hyvyyttä arvioidaan Connect-pelissä.

- *Satunnainen pelaaja* valitsee satunnaisen laillisen siirron
- *Naive* valitsee satunnaisen siirron, paitsi kahdessa poikkeustapauksessa. 1) Jos löytyy jokin samanlaisen nappulan suora, jonka pää voidaan täydentää jommankumman pelaajan seuraavalla vuorolla muodostaen voittosuora. Tällöin Naive lisää nappulan suoran päähän riippumatta kumman pelaajan suora on; joko se voittaa tai estää voittosiirron. 2) Naive ei tee siirtoa joka johtaisi välittömästi vastustajan avoimeen voittavaan suoraan. [Sch02]
- *minimax-9-pelaaja* käy siirtoja läpi maksimissaan yhdeksän siirtoa eteenpäin ja katsoo, mikä niistä johtaa parhaaseen tulokseen. Jos optimaalisia tuloksia löytyy useita, se valitsee niistä jonkin satunnaisesti. minimax-9:ää käytetään vain 3x3-laudalla, missä sitä ei voi voittaa.
- Rinnakkaisevoluution ja jonkin PSO-variaation yhdistelmällä kehitettyä tekoälyä soveltava pelaaja, joka valitsee neuroverkon mukaan parhaaseen lauta-asetelmaan johtavan siirron. Pelaaja käyttää minimax-algoritmia pelipuun läpikäymiseen vain yhden siirron syvyyteen, jolloin neuroverkon kehittyneisyyttä testataan äärimmäisesti ja säästetään laskenta-aikaa. Tätä toimintamallia käytetään sekä kehitys- että vertailuvaiheessa. Käytetyt PSO-menetelmät ovat seuraavat:
    - Perus-PSO (luku 4.1) – antaa yleiskuvan parvioptimoinnin suorituskyvystä
    - FIPS (luku 4.3) – perusversiota parempi PSO kertoo, miten hyviä tuloksia on ainakin mahdollista saavuttaa
    - CLPSO (luku 5.9) – toinen perusversiota kehittyneempi tekniikka

6.5.3 Testaustapa ja käytettyjä arvoja

Connect-peliä testatataan eri kokoisilla laudoilla. 3x3-pelilaudalla voittoon tarvitaan 3 peräkkäistä siirtoa. Jos taas vapaita ruutuja ei ole eikä peräkkäisiä siirtoja ole tarpeeksi, on kyseessä tasapeli. 4x4-laudalla peräkkäisiä siirtoja tarvittiin voittoon neljä (4) kappaletta ja 7x6-laudalla samoin neljä (4).

Pelaajan staattista evaluointifunktiota kehitetään rinnakkaisevoluution ja PSO-menetelmien avulla testitapauksen mukainen iteraatiomäärä. Sen jälkeen kehitettyä tekoälyä soveltava

pelaaja pelaa jotakin referenssivastustajaa vastaan 10 000 kertaa – poikkeuksena minimax-9-vastustaja, jota vastaan pelataan kehittämisen jälkeen 100 kertaa. Perusteluna vain 100:lle kerralle ovat laskentataakan väheneminen ja minimaxin optimaalinen toiminta: se valitsee satunnaisesti vain, jos on olemassa useita optimaalisia siirtoja. Ilman satunnaisuutta minimaxin ja kehitetyn tekoälyn pelit päättyisivät aina samoin, mikä antaisi melko rajoitetun kuvan tekoälyn hyvyydestä. Tilastollisesti kelvollisempien tulosten laskemiseksi tarvittaisiin kuitenkin enemmän kuin 100 peliä testiä kohden.

Rinnakkaisevoluutiossa jokaista pelaajaa peluutetaan valittu määrä satunnaisia pelaajia vastaan. Koska tälla tavalla jotkin pelaajat pelaavat mahdollisesti enemmän pelejä, pelaajan pistemäärä on suhteellinen pelimäärään. Voitosta saa 3 ja tasapelistä 2 pistettä – häviöstä ei mitään. Lisäksi kaikissa paitsi yhdessä testissä (3) kehitetään 1. ja 2. pelaaja erillisessä PSO-systeemissä.

Käytetyt PSO-tekniikat ovat Basic eli perus-PSO, CLPSO ja FIPS. Menetelmissä käytetyt parametrit olivat:

Basic:	cMax = 1,47, c1 = 0,7
CLPSO:	m = 7, c = 1,49445, w_0 = 0,9, w_1 = 0,4
FIPS:	cMax = 1,47, c1 = 0,7

Basic käyttää gbest-naapurustomallia, eli tieto leviää välittömästi kaikille. FIPSissä naapurusto muodostetetaan listaamalla partikkelit ja valitsemalla naapureiksi listan vasemmalla ja oikealla puolella oleva partikkeli itsensä lisäksi. CLPSO-menetelmässä iteraatiosidonnainen kerroin *w* ja *i*:nnen partikkelin Pc_i saadaan kaavoista:

$$w(t) = w_1 + (w_0 - w_1) \cdot \frac{iteraatioita - t}{iteraatioita}$$

$$Pc_i = 0{,}05 + 0{,}45 \cdot \frac{e^{\frac{10(i-1)}{ps-1}} - 1}{e^{10} - 1},$$

missä *t* kertoo, monesko iteraatio on menossa, *ps* on partikkelien määrä ja *e* on neperin luku. [Lia06]

Testiohjelma on toteutettu Scala-ohjelmointikielellä, ja ohjelma ajettiin AMD Athlon 2600 XP -tietokoneessa, jossa muistia on 1024 Mt. Testit toistetaan 30 kertaa samoille parametreille ja tekoälyä soveltavan pelaajan voitto-, tasapeli- ja häviömääristä otetaan keskiarvo ja standardipoikkeama.

6.5.4 Testitapaukset

3x3

Testi 1. Referenssipelaajien keskinäinen tasoero

Tarkoituksena on selvittää referenssipelaajien keskinäinen ero, jotta verrattaessa kehitettyjä tekoälypelaajia niihin saadaan kattava kuva käytetyn menetelmän toimivuudesta. Logiikkaa sisältävän Naive-pelaajan pitäisi voittaa satunnaispelaaja selvästi.

Testi 2. Perus-, FIPS- ja CLPSO-menetelmien keskinäinen tasoero tekoälypelaajan pelatessa minimax-9:ää vastaan

Halutaan selvittää, miten testattavat PSO-tekniikat eroavat toimintakyvykkyydessään toisiinsa verratessa. Myös parven koon vaikutusta testataan. Kehittyneempien tekniikoiden (FIPS ja CLPSO) odotetaan toimivan perus-PSO:ta paremmin tutkielmassa esitettyjen tietojen perusteella.

Testi 3. Pelaajan kehittäminen a) samalla b) eri PSO:lla ykkös- ja kakkospelaajaksi

Halutaan selvittää, onko kannattavaa kehittää tekoälypelaaja samalla kertaa sekä ykkös- että kakkospelaajaksi käyttäen yhtä PSO:ta, vai kannattaako pelaajat kehittää toisistaan riippumattomasti kahdella PSO:lla – toinen tekoälypelaaja ykkös- ja toinen kakkospelaajaksi. Menetelmien välillä ei pitäisi olla suurta eroa.

Testi 4. PSO-iteraatiomäärän vaikutus CLPSO:n kehittämän pelaajan suoriutumiseen

Halutaan tutkia, kuinka paljon positiivista vaikutusta on PSO-iteraatiomäärän lisäämisellä CLPSO-menetelmässä. Iteraatiomäärän pitäisi vaikuttaa selvästi tulokseen aiempien menetelmään liittyvien tutkimustulosten perusteella [Mes06]. Lisäksi selvitetään, kuinka suuri vaikutus rinnakkaisevoluutiossa käytettävien vastustajien lukumäärällä on; etukäteen selvää on, että mitä isompi lukumäärä on, sitä tarkempi

pelaajien suhteellinen evaluointi on. Eroa viiden (5) ja kymmenen (10) vastustajan käytöllä pitäisi olla merkittävä.

Testi 5. Iteraatiomäärän vaikutus FIPS -menetelmällä kehitetyn pelaajan suoriutumiseen

Halutaan selvittää, kuinka paljon positiivista vaikutusta on PSO-iteraatiomäärän lisäämisellä FIPS-menetelmässä. Iteraatiomäärän pitäisi vaikuttaa selvästi tulokseen aiempien menetelmään liittyvien tutkimustulosten nojalla [Mes06].

4x4

Testi 6. Referenssipelaajien keskinäinen tasoero

Selvitetään referenssipelaajien välinen suoriutumiskyky 4x4-pelilaudalla, jotta kehitettyä tekoälypelaajaa voidaan arvioida kattavasti. Naive-pelaajan pitäisi voittaa selvästi satunnaispelaaja.

Testi 7. Parven koon vaikutus FIPS-menetelmällä kehitetyn tekoälypelaajan suoriutumiseen

Halutaan saada selville, kuinka suuri vaikutus parven koolla on FIPS-menetelmässä, kun kehitetään tekoälypelaajaa. Testi antanee myös tietoa menetelmän yleisestä toimivuudesta 4x4-laudalla, jossa on haastavampaa menestyä kuin 3x3-laudalla.

Parven koolla ei kirjallisuuskatsauksen perusteella pitäisi olla suurta vaikutusta.

7x6

Testi 8. Referenssipelaajien keskinäinen tasoero

Selvitetään referenssipelaajien välinen suoriutumiskyky 7x6-pelilaudalla, jotta voidaan arvioida kehitetyn tekoälypelaajan suorituskykyä kattavasti. Naive-pelaajan pitäisi voittaa selvästi satunnaispelaaja.

Testi 9. FIPS-menetelmällä kehitetyn pelaajan suoriutuminen Naive-pelaajaa vastaan

Testataan, kuinka hyvin FIPS-menetelmällä kehitetty tekoälypelaaja suoriutuu 7x6-pelilaudalla. Tämä siitä syystä, että 7x6-laudalla pelattaessa siirtovariaatioita on huomattavasti enemmän kuin 3x3- tai 4x4-laudalla, ja näin testi antanee kattavan kuvan käytetyn menetelmän toimivuudesta.

Kehitetyn tekoälypelaajan odotetaan suoriutuvan kohtuullisesti Naive-pelaajaa vastaan; toimivuuden ennalta arvioiminen on kuitenkin vaikeaa.

6.6 Tulokset

6.6.1 3x3

Seuraavassa 3x3-laudalla saadut tulokset.

Testi 1. Referenssipelaajien keskinäinen tasoero

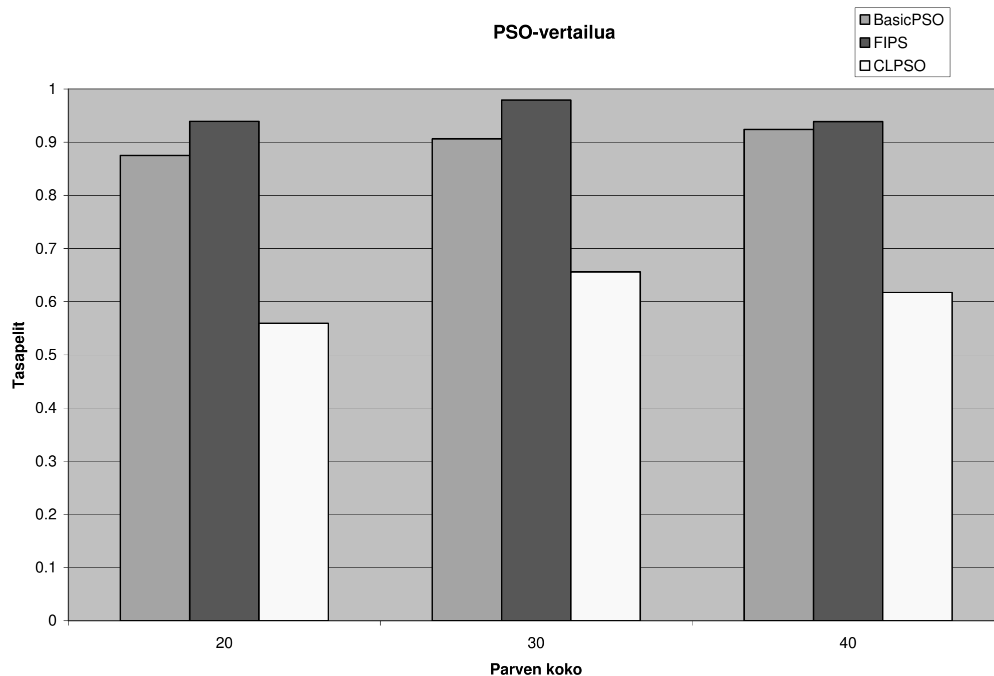
Testattiin referenssinä käytettyjen tekoälyjen keskinäistä eroa 3x3-laudalla (Liite 1: Kuva 14).

Satunnaispelaaja (random) voittaa yli puolet peleistään itseään vastaan. Tasapelejä tulee niukasti alle 20 %:a. Naive pelaa itseään vastaan hieman paremmin kakkospelaajana, mikä ei ollut etukäteen odotettavissa; tasapelejä tulee kuitenkin eniten n. 40 %:a.

Naive voittaa satunnaispelaajan selvästi sekä ykkös- että kakkospelaajana. Ykköspelaajana se voittaa n. 70 %:a, ja tasapelejä tulee n. 20 %:a. Kakkospelaajana vastaavat luvut ovat n. 45 % ja 33 %.

Testi 2. Perus-, FIPS- ja CLPSO-menetelmien keskinäinen tasoero tekoälypelaajan pelatessa minimax-9:ää vastaan

Basic:a, FIPS:ia ja CLPSO:ta verrattiin minimax-9:ää vastaan, joka ei häviä koskaan 3x3-laudalla. Tekoälysovelletusta kehitettiin 300 iteraatiota ja kokeiltiin 20, 30 ja 40:llä partikkelilla. Piiloneuroneita oli viisi (5) ja rinnakkaisevoluutiovastustajia viisi (5).



Kuva 11. PSO-tekniikoiden pelaamien tasapelien osuus minimax-9:ää vastaan käydyissä otteluissa.

Kuva 11 näyttää tasapelien osuuden PSO-tekniikoiden kehittämän pelaajan ollessa ykköspelaajana. FIPS on kaikilla iteraatiomäärillä paras yli 90 %:n tasapelimäärällä. Perus-PSO suoriutuu muutamia prosenttiyksikköjä heikommin. CLPSO sen sijaan on selvästi huonoin; parhaimmillaankin se saavuttaa 66 %:n tasapeliosuuden. Lisäksi parven koon vaikutus ei ole merkittävä.

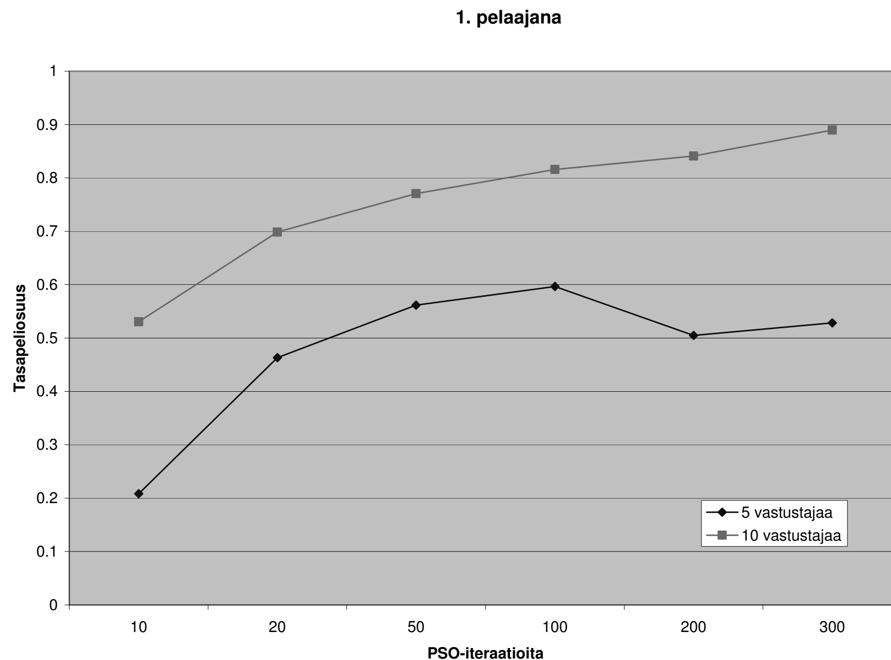
Testi 3. Pelaajan kehittäminen a) samalla b) eri PSO:lla ykkös- ja kakkospelaajaksi

FIPS:lla kehitettiin pelaajaa 200 iteraatiota käyttäen 20 partikkelia, viittä (5) piiloneuronia ja viittä (5) rinnakkaisevoluutiovastustajaa, ja pelaajan referenssivastustajana oli Naive. Kehitystä kokeiltiin kahdella tavalla: ensimmäisessä 1. ja 2. pelaaja kehitettiin erillisillä ja toisessa samalla PSO:lla. Jälkimmäisessä tapauksessa evaluointifunktiossa käytetään siis samaa neuroverkkoa sekä 1. että 2. pelaajana.

Erikseenkehitysmenetelmä saavuttaa kokonaisuudessaan paremmat tulokset sekä 1. että 2. pelaajana (Liite 1: Kuva 15) Yhdellä PSO:lla kehittäminen on ykköspelaajaa tarkastellessa hieman parempi, mutta tasapelien osuus on sitä vastoin huonompi. Kakkospelaajana eroa menetelmien välillä on n. neljän (4) prosenttiyksikön verran tasapeleissä ja kahdeksan (8) verran voitto-osuudessa.

Testi 4. PSO-iteraatiomäärän vaikutus CLPSO:n kehittämän pelaajan suoriutumiseen

CLPSO:lla kehitettiin pelaajaa 10, 20, 50, 100, 200 ja 300 iteraatiota ja testattiin tätä minimax-9:ää vastaan (100 peliä per kerta). Parven koko oli 20 ja piiloneuroneita viisi (5). Rinnakkaisevoluutiotestivastustajamäärät olivat viisi (5) ja kymmenen (10).



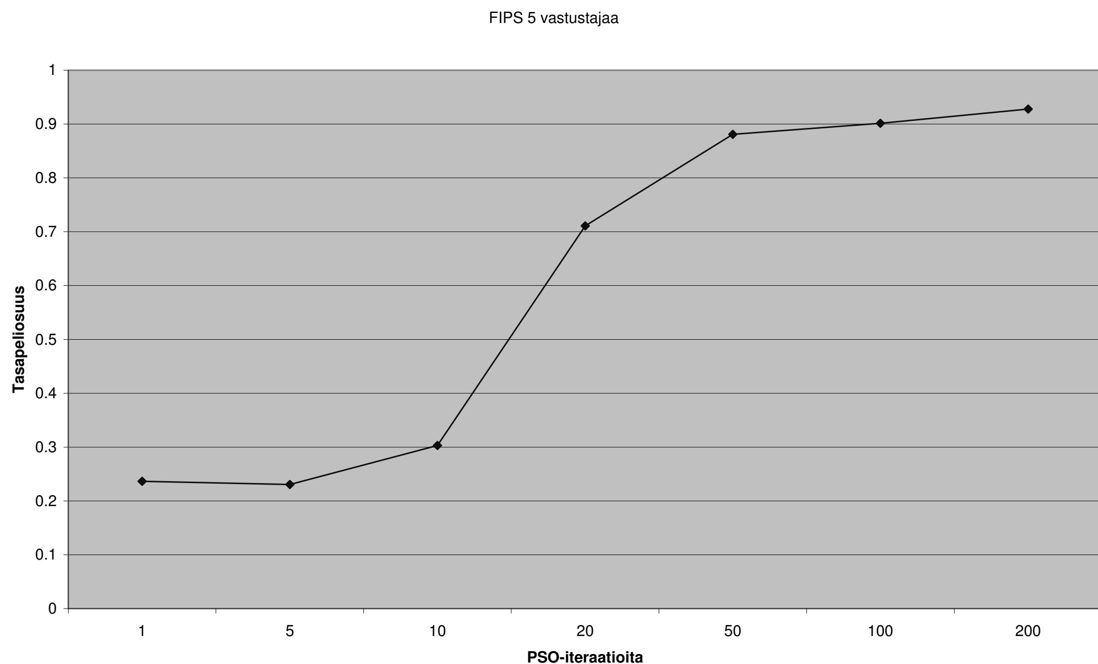
Kuva 12. CLPSO-pelaajan kehittyminen ykköspelaajana tasapelien kannalta iteraatiokertojen funktiona. Vastustajana on minimax-9. Rinnakkaisevoluutiossa käytettäviä vastustajia on 5 ja 10.

Iteraatiomäärän kasvattamisella on selvä vaikutus. Kuitenkin viidellä vastustajalla 50:n iteraation jälkeen tasapeliprosentti pysyttelee 50–60 välillä. Kymmenellä vastustajalla 50:n iteraation jälkeen tasapeliprosentti on 77 ja 300:n iteraation jälkeen se on 89. (Kuva 12)

Kakkospelaajana saavutetaan jo pienillä iteraatioilla korkeita prosentteja, mutta suuremmilla iteraatiomäärillä saadaan parannettua tulosta muutamia prosenttiyksiköitä – kymmenellä vastustajalla yli 90 %:iin. (Liite 1: Kuva 16)

Testi 5. Iteraatiomäärän vaikutus FIPS-menetelmällä kehitetyn pelaajan suoriutumiseen

FIPS-pohjaista tekoälyä kehitettiin 1, 5, 10, 20, 50, 100 ja 200 iteraatiota, kun rinnakkaisevoluutiovastustajamäärä oli viisi (5) ja piiloneuroneita viisi (5). Kehittämisen jälkeen tekoälypelaaja pelasi minimax-9:ää vastaan 100 kertaa.



Kuva 13. FIPS-menetelmällä kehitettyjen pelaajien tasapeliosuus PSO-iteraatioiden funktiona pelattaessa ykköspelaajana minimax-9:ää vastaan.

Ykköspelaajana jo muutaman kymmenen lisäiteraation lisäys nostaa tasapelien osuutta selvästi: 20:lla iteraatiolla 71 %:a peleistä päättyy tasapeliin ja 50:llä 88 %:a. (Kuva 13)

Kakkospelaajana FIPS saadaan 200:lla iteraatiolla 96 %:iin (Liite 1: Kuva 17). Ero CLPSO:hon on pääosin 10 prosenttiyksikön luokkaa eri iteraatiomäärillä.

6.6.2 4x4

Seuraavassa 4x4-laudalla saadut tulokset.

Testi 6. Referenssipelaajien keskinäinen tasoero

Voittoja satunnaispelaajalle tuli jonkin verran enemmän ykköspelaajana (28 %:a) kuin kakkospelaajana (24 %:a). Tasapelejä tuli lähes puolet (48 %:a). Naiven pelaaminen itseään vastaan tuotti tasapelejä 79 %:a ja voittoja hieman enemmän kaksinpelaajana (12 %:a) kuin yksinpelaajana (8 %:a). (Liite 1: Kuva 18)

Tasapelejä tuli paljon myös Naiven ja satunnaispelaajan välillä; Naiven ollessa ensimmäinen pelaaja voittoja tuli 45 %:a ja tasapelejä 49 %:a. Kaksinpelaajana Naive voitti 39 %:a peleistä ja 56 %:a päättyi tasapeliin.

Testi 7. Parven koon vaikutus FIPS-menetelmällä kehitetyn tekoälypelaajan suoriutumiseen

FIPSiä kehitettiin 100 iteraatiota arvoilla: rinnakkaisevoluutiovastustajat viisi (5), piiloneuronit viisi (5). Testatut parven koot olivat 10, 20, 30, 40 ja 50. Tekoälyä testattiin satunnaispelaajaa vastaan.

Ykköspelaajana parven kokoa 10:stä 50:een nostamalla saatiin voittosuhde 55 %:sta 60 %:iin (Liite 1: Kuva 19). Kakkospelaajana parven koon vaikutus oli samaa luokkaa kuin ykköspelaajana, prosenttiosuus parani 40:stä 45:een. 45 %:a saavutettiin parven koolla 30 eikä nosto 40:een tai 50:een enää tuonut lisähyötyä (Liite 1: Kuva 20).

6.6.3 7x6

Seuraavassa 7x6-laudalla saadut tulokset.

Testi 8. Referenssipelaajien keskinäinen tasoero

Tasapelejä tuli selvästi alle prosentti kaikissa paitsi Naiven itseään vastaan (9 %:a) pelatuissa peleissä. Naive voitti 48 %:a ykköspelaajana ja 43 %:a kakkosena itseään vastaan. Satunnaispelaaja voitti itseään vastaan 55 %:a ykköspelaajana ja 44 %:a kakkosena. (Liite 1: Kuva 21)

Ykköspelaajana Naive voitti satunnaispelaajan 95 %:a peleistä, kun taas kakkospelaajana 92 %:a. Sangen pieni osa peleistä päättyi tasapeliin, ja loput voitti satunnaispelaaja – ykköspelaajana 8 %:a ja kakkospelaajana 5 %:a.

Testi 9. FIPS-menetelmällä kehitetyn pelaajan suoriutuminen Naive-pelaajaa vastaan

FIPSiä kehitettiin 100:a ja 200:aa iteraatiota. Parven koko oli 20, rinnakkaisevoluutiovastustajia viisi (5) ja piiloneuroneita viisi (5). Kehitettyä tekoälyä testattiin Naive-pelaajaa vastaan.

Ykköspelaajana 100:lla ja 200:lla iteraatiolla ei ollut juuri vaikutusta: saavutettu voittoprosentti oli 33. Kakkospelaajana voittoja tuli 100:lla iteraatiolla 18 %:a ja 200:lla 21 %:a. Tasapelejä tuli kaikissa tapauksissa korkeintaan 1,5 %:a. (Liite 1: Kuva 22)

6.7 Tulosten analysointi

6.7.1 Yleistä

Seuraavaksi tarkastellaan saatuja tuloksia eri PSO-tekniikoiden välillä, ja pohditaan kuinka eri parametrit vaikuttavat tekoälyn oppimiseen. Voittojen, tasapelien ja häviöiden määrää analysoitaessa on huomioitava, että parasta tekoälypelaajaa etsittäessä painotettiin voittoja enemmän kuin tasapelejä: pelaajan hyvyyden määräävät pisteet laskettiin kaavalla $3 \cdot voitot + 2 \cdot tasapelit$.

6.7.2 3x3

Testi 1. Referenssipelaajien keskinäinen tasoero.

Referenssipelaajista satunnaispelaajan ollessa ykköspelaajana tämä voittaa itseään vastaan yli puolet peleistä. Tässä tapauksessa pelaajat itseasiassa tekevät siirtonsa ymmärtämättä siirron merkitystä. Yksinkertaisen logiikan lisääminen (Naive) nosti ykköspelaajan voittoprosentin yli 65:een. Kakkospelaajana satunnaiset siirrot johtivat alle 30 %:n voittoihin, kun taas Naive voitti tällöin satunnaispelaajan 45 %:sti. Tasapelejäkin tuli yli 30 %:a eli yli 10 prosenttiyksikköä enemmän kuin satunnaisen pelatessa itseään vastaan. (Liite 1: Kuva 14)

Minimax-9:llä ei pelattu muita referenssipelaajia kuin itseään vastaan. Näin siitä syystä, että tiedetään ilmankin sen voittavan tai pelaavan tasan aina oli vastus mikä tahansa. Itseään vastaan se saavuttaa 100 %:sti tasapelejä.

Testi 2. Perus-, FIPS- ja CLPSO-menetelmien keskinäinen tasoero tekoälypelaajan pelatessa minimax-9:ää vastaan

Minimax-9:ää vastaan parven koko ei juuri vaikuttanut minkään PSO-tekniikan tuloksiin (Kuva 11). Yksi syy siihen lienee suhteellinen ja epätarkka evaluointimenetelmä, sillä vaikka suuremmalla parvella saadaan kattavampi haku, niin pelaajien keskinäisen paremmuuden arvionti on entistä vaikeampaa.

PSO-menetelmistä FIPS toimii perus-PSO:ta paremmin kaikilla parvikokomäärillä. Poikkeavaa sen sijaan on CLPSO:n heikko menestyminen. Se jää jopa yli 30 prosenttiyksikköä huonommaksi kuin FIPS. Tulos mahdollisesti kertoo siitä, että CLPSO:lla

on vaikeuksia konvergoitua ympäristössä, jossa arvoja evaluoidaan suhteellisesti ja epätarkasti.

Testi 3. Pelaajan kehittäminen a) samalla b) eri PSO:lla ykkös- ja kakkospelaajaksi

Ykkös- ja kakkospelaajaa kehitettiin sekä samanaikaisesti yhdellä PSO:lla että erikseen kahdella PSO:lla. Erikseen kehitettyinä tekoälypelaajat suoriutuvat jonkin verran paremmin – kakkospelaajana ero on jopa 10 prosenttiyksikköä (Liite 1: Kuva 15). Lisätutkimusta tarvitaan eri parametreilla, jotta voitaisiin arvioida, pitääkö tämä yleisemmin paikkansa.

On huomioitava, että erikseen kehitettynä laskenta-aika ei suosi yleisesti toista tekniikkaa; silloin, kun testattavalle tekoälylle valitaan testivastustajat, pelaavat ne ainoastaan omaa pelirooliaan – 1. tai 2. pelaajana, ei molempina. Samanaikaisesti 1. ja 2. pelaajaa kehittäessä sen sijaan joudutaan pelaamaan sekä ykkös-, että kakkospelaajana, jotta ykköspelaajaksi valittu ei saa etulyöntiasemaa (todennäköisempää voittaa). Toisessa tekniikassa tarvitsee siis vähemmän PSO-kehitystä ja toisessa rinnakkaisevoluutiopelimääriä.

Testi 4. PSO-iteraatiomäärän vaikutus CLPSO:n kehittämän pelaajan suoriutumiseen

CLPSO:lla kehitettiin 10:stä 300:aan iteraatiomäärää ja aikaansaatua pelaajaa testattiin minimax-9:ää vastaan. Iteraatiomäärän nostaminen kasvatti tasapelimäärää varsin kohtuullisesti ykköspelaajana pelatessa (Kuva 12). Viidellä rinnakkaisevoluutiovastustajalla iteraatiomäärän kasvattaminen ei tuonut kuitenkaan selvää etua enää 100:n iteraation jälkeen; sillä oli jopa negatiivinen vaikutus. Kuitenkin rinnakkaisevoluutiovastustajien määrän lisäys kymmeneen paransi iteraatiomäärän kasvun vaikutusta vakaammaksi ja suuremmaksi; tällöin CLPSO suoriutui varsin hyvin. Tästä voidaan päätellä, että CLPSO on herkkä rinnakkaisevoluutiossa käytettävän evaluoinnin tarkkuudelle ja että on järkevää käyttää useampaa kuin viittä (5) vastustajaa. Kakkospelaajana tulokset olivat myös oikein hyvät (Liite 1: Kuva 16), mutta tässä tapauksessa minimax-9:ää vastaan näyttäisi olevan varsin helppoa saada tasapeli aikaan muutenkin.

Testi 5. Iteraatiomäärän vaikutus FIPS -menetelmällä kehitetyn pelaajan suoriutumiseen

FIPS:llä kehitettiin pelaajaa väliin 1–200 kuuluvilla iteraatiomäärillä. On huomattavaa, miten nopeasti FIPS kehittyy jo yli 90 %:n tasapeliosuuteen sekä ykkös- että kakkospelaajana (Kuva 13 ja Liite 1: Kuva 17). Tämän testin mukaan FIPS konvergoituu

optimiin huomattavasti CLPSO:ta paremmin, kun käytetään viittä (5) rinnakkaisevoluutiovastustajaa.

6.7.3 4x4

Testi 6. Referenssipelaajien keskinäinen tasoero

Referenssipelaajien keskinäisistä otteluista on huomattavaa, että tasapelejä tuli varsin paljon (Liite 1: Kuva 18). Etenkin Naive pelasi lähes 80 %:a peleistään tasan itseään vastaan. Satunnaispelaajaa vastaan Naive voitti hieman harvemmin kuin mitä pelasi tasan, sekä ykkös- että kakkospelaajana. Selvästi Naive on kuitenkin parempi.

Testi 7. Parven koon vaikutus FIPS-menetelmällä kehitetyn tekoälypelaajan suoriutumiseen

FIPSillä testattiin parven koon vaikutusta kehitykseen 100:n iteraation jälkeen. Vaikutus oli varsin pieni, n. 5 prosenttiyksikköä (Liite 1: Kuva 19 ja Kuva 20). Satunnaispelaajaa vastaan voittoja tuli yli 55 %:a ykköspelaajana ja yli 40 %:a kakkospelaajana, mikä on hieman paremmin kuin se, miten Naive pelasi satunnaispelaajaa vastaan. Tämänkin testin perusteella on syytä olettaa, että tarvitaan enemmän rinnakkaisevoluutiossa käytettäviä vastustajia, jotta suhteellinen evaluointi olisi tarkempaa erityisesti suurilla parvilla.

6.7.4 7x6

Testi 8. Referenssipelaajien keskinäinen tasoero

Referenssipelaajilla tuli tasapelejä erittäin niukasti (Liite 1: Kuva 21). Naiven pelatessa itseään vastaan niitä tuli kuitenkin melkein 10 %:a. Naive voitti satunnaispelaajan ylivoimaisesti (yli 90 %:a) sekä ykkös- että kakkospelaajana. Tulos kyseenalaistaa sitä, että tämän tutkimuksen pohjalla oleva työ [Mes04] käyttää satunnaispelaajaa verrokkivastustajana; se antaa mahdollisesti liian optimistisen kuvan kehitetyn älyn hyvyydestä.

Testi 9. FIPS-menetelmällä kehitetyn pelaajan suoriutuminen Naive-pelaajaa vastaan

FIPS:llä kehitettiin tekoälypelaajaa, ja se pelasi Naive-pelaajaa vastaan, kun kehitysiteraatioita oli 100 ja 200. Iteraatiomäärä ei oleellisesti vaikuttanut tulokseen; ykköspelaajana FIPS voitti n. 33 %:a peleistä ja kakkospelaajana n. 20 % tapauksista. (Liite 1: Kuva 22) Tulos ei ole erityisen hyvä, sillä Naive voitti itseään vastaan 48 %:a

ykköspelaajana ja 43 %:a kakkospelaajana. Verrattuna satunnaispelaajaan FIPS kuitenkin suoriutui hyvin, parikymmentä prosenttiyksiköitä paremmin.

6.8 Yhteenveto

Tuloksien perusteella pelaajan kehittäminen on mahdollista rinnakkaisevoluution ja PSO:n yhdistelmällä. FIPS on lähes kauttaaltaan paras PSO-tekniikka testatuista menetelmistä. Iteraatioiden määrä vaikuttaa selvästi sovelletuksen laatuun, ja 1. ja 2. pelaajien kehittäminen erikseen on toimiva idea. Parven koolla sen sijaan ei ollut juuri vaikutusta tuloksiin, mikä johtunee osittain epätarkasta ja suhteellisesta evaluoinnista.

3x3-laudalla tekoälypelaajat suoriutuvat erittäin hyvin, 4x4:llä kohtuullisesti ja 7x6:lla myönteistä kehitystä on selvästi havaittavissa. Tarvittaisiin kuitenkin enemmän testejä ja suurempia iteraatiomääriä, jotta voitaisiin päätellä, kuinka hyvin käytetty menetelmä toimii etenkin 7x6-tapauksessa.

Koska käytetyt iteraatiomäärät olivat melko alhaisia, ei konvergoitumista ehtinyt tapahtua useinkaan, ja sen vuoksi tuloksissa on kohtuullisen suuri standardipoikkeama (usein 10 %:a pelimäärästä). Lisäksi, koska keskiarvo otettiin 30:stä ajokerrasta, tuloksiin on syytä suhtautua lähinnä suuntaa antavasti. Johdonmukainen parempien tulosten saaminen suuremmilla iteraatiomäärillä ja FIPSin yleinen suoriutuminen suhteessa perus-PSO:hon kertovat kuitenkin tulosten järkevyydestä.

7 Pohdinta

Tässä työssä esitetyt kolme erilaista parviälytekniikkaa – muurahaisoptimointi (ACO), stokastinen diffuusiohaku (SDS) ja parvioptimointi (PSO) – ovat hyviä vaihtoehtoja lukuisiin optimointiongelmiin. Erityisesti muurahaisoptimoinnilla luonnistuu kombinatoristen ongelmien (likimääräinen) ratkaisu ja parvioptimoinnilla minkä tahansa funktion optimointi useaulotteisessa reaalilukuavaruudessa. Stokastinen diffuusiohaku on puolestaan käyttökelpoinen hahmonsovituksessa.

Yleisesti ottaen parviäly on käyttökelpoinen tekniikka lukuisilla sovellusalueilla. Esimerkiksi robottien kehityksessä sen etuina ovat toiminnallisuuden skaalautuvuus ja valmistuksen edullisuus, sillä yksilöt ovat toiminnaltaan yksinkertaisia ja vuorovaikutus tapahtuu *stigmergyn* eli ympäristön muokkauksen avulla. [Bon09, Her06] Yksilön hakuälyyn liittyvä prosessointikapasiteetti voi olla myös varsin pieni, sillä agenttien tarvitsee pääosin laskea uusi sijainti yksinkertaisella kaavalla. Parven toimintakykykin on joustava, sillä yhden robotin särkyminen ei oleellisesti vaikuta parven toimintaan. [Her06]

Parvioptimointia käsiteltiin laajasti, sillä sitä on tutkittu varsin runsaasti viime vuosina. Kehitetyt parvioptimointitekniikat pyrkivät toimimaan usein tiettyjen vaatimusten suhteen. Esimerkiksi Clercin TRIBES on tehokas adaptiivinen tekniikka, joka mukautuu ympäristöön eikä sille tarvitse määritellä etukäteen juuri mitään parametreja. [Cl06a] Esille tuotiin myös parvioptimoinnin analysointia, joka on haastavaa stokastisen luonteen vuoksi. Lisäksi uusien tekniikoiden kehittäminen ja vertaaminen aiempiin on vaativaa reaalimaailman ongelmien kompleksisuuden vuoksi. Standarditestiaineisto antanee kuitenkin viitettä siitä, miten hyviä eri tekniikat ovat. NFL-teoreema puolestaan kertoo, ettei voida kehittää optimointimenetelmää, joka toimisi parhaiten kaikille tapauksille [Wol96]. Toisaalta, jos optimoitava funktio käyttäytyy ennakoitavasti, esimerkiksi kuvaamalla läheiset arvot lähekkäin, on optimointimetodien välistä hyvyyttä mahdollista ja suotavaakin arvioida kyseisessä ongelmaluokassa [Chr01].

Tässä työssä kokeiltiin rinnakkaisevoluution ja PSO:n yhdistämistä tekoälyn kehittämisessä Connect-peliin; vastaavaa on kokeiltu muille peleille [Mes06]. Ideana on, että PSO:n avulla haetaan optimaalisia painoja neuroverkkoon, jota tekoälypelaajan evaluointifunktio käyttää. Pelaajat pelaavat vastakkain käyttäen omaa evaluointifunktiotaan, ja ottelutuloksista saadut

pisteet vaikuttavat tekoälyjen suhteelliseen hyvyyteen, mikä puolestaan on PSO:ssa käytettävän evaluointifunktion arvo. [Mes04] Tätä menetelmää testattiin eri parametreilla etenkin 3x3-laudalla, ja vaikka laskenta-aikaa olisi tarvittu enemmän tilastollisesti merkittävien tulosten aikaansaamiseksi, nähdään jo saaduilla tuloksilla selvästi parametrien vaikutustrendejä.

Kehitetyt tekoälysovellukset selviytyivät satunnaispelajaa vastaan erittäin hyvin. Suuremmalla kuin 3x3-lautakoolla taso oli suunnilleen sama kuin Naive-pelaajalla. 3x3-laudalla joillakin parametreilla saatiin keskimäärin minimax-9:ää vastaan erittäin hyvin suoriutuva pelaaja. Odottamatonta PSO-tekniikoiden keskinäisessä suoriutumisessa on CLPSO:n heikko menestys suhteessa FIPSiin. Syy tähän lienee CLPSO:n toimintastrategian soveltumattomuus siihen, että tekoälyjen suhteellisen hyvyyden määrääminen on melko epätarkkaa viidellä (5) rinnakkaisevoluutiovastustajalla; useammalla (10) vastustajalla CLPSO:n tulos parani selvästi. Tätä olisi syytä tutkia tarkemmin.

Avoimeksi kysymykseksi jää, kannattaako ykkös- ja kakkospelaajia kehittää samassa vai kahdessa erillisessä PSO:ssa. Erikseenkehitys vaikuttaa järkevältä menetelmältä, ja sitä käytettiinkin lähes kaikissa testeissä – toisin kuin artikkelissa, jossa rinnakkaisevoluution ja PSO:n yhdistäminen esiteltiin [Mes04]. Lisäksi mielenkiintoista olisi tietää, kannattaako rinnakkaisevoluutiossa käytettävä määrä vastapelaajia valita satunnaisesti jokaiselle pelaajalle erikseen, vaiko valita samat satunnaisvastustajat kaikille.

Iteraatiomäärän positiivinen vaikutus on odotettua. 3x3-laudalla se on selvästi nähtävissä. 4x4- ja 7x6-laudalla tehdyt testit olivat sen sijaan liian suppeita parametrien arvojen vaikutuksen arviointiin. Tosin parven koolla ei näyttäisi olevan juuri merkitystä, ainakaan kun käytetään evaluoinnissa vain muutamia rinnakkaisevoluutiovastustajia. Joka tapauksessa olisi toivottavaa selvittää, miten FIPS suoriutuu usean tuhannen iteraation jälkeen. Piiloneuronien merkitys voi myös korostua isoilla laudoilla, sillä tällöin edes tasapeleihin yltäminen ei onnistu helposti testien mukaan. Näin ollen on mahdollista, että neuroverkon painojen kytkentöjä tarvitaan enemmän hahmojen tunnistamiseen.

Viitteet

[All88] Allis, V. (1988). A Knowledge-based Approach of Connect-Four [Pro gradu]. Vrije Universiteit, Amsterdam.

[Alt06] Altshuler, Y., Yanovsky, V., Wagner, I.A. & Bruckstein, A.M. (2006). Swarm Intelligence Systems (Swarm Intelligence - Searchers, Cleaners and Hunters). *Studies in Computational Intelligence*, vol. 26, Springer Verlag.

[Bir06] Bird, S. & Li, X. (2006). Adaptively Choosing Niching Parameters in a PSO. *Proceedings of the 8th annual conference on Genetic and evolutionary computation*, 3–10.

[Bis89] Bishop, J. M. (1989). Stochastic Searching Networks. *Proc. 1st IEE Conf. on Artifical Neural Networks*, 329–331.

[Bla02] Blackwell, T. M. & Bentley, P. J. (2002). Dynamic Search with Charged Swarms. *Proceedings of the Genetic and Evolutionary Computation Conference*, 19–26.

[Bon99] Bonabeau, E., Dorigo, M. & Theraulaz, G. (1999). *Swarm Intelligence: From Natural to Artificial Systems*. Oxford University Press, New York.

[Chr01] Christensen, S. & Oppacher, F. (2001). What can we learn from No Free Lunch? A first attempt to characterize the concept of a searchable function. *Proc. Genetic and Evolutionary Computation Conference GECCO-2001*. Morgan Kaufmann, San Francisco, CA, 2001. 1219–1226.

[Cl06a] Clerc, M. (2006). *Particle Swarm Optimization*. ISTE Ltd, London, UK.

[Cl06b] Clerc, M. (2006). Stagnation analysis in particle swarm optimisation or what happens when nothing happens. *Technical Report CSM-460*, Department of Computer Science, University of Essex.
http://clerc.maurice.free.fr/pso/stagnation_analysis/Stagnation_Analysis_csm-460.pdf (tarkistettu: 12.11.2007).

[Col92] Colorni, A., Dorigo, M. & Maniezzo V. (1992). Distributed Optimization by Ant Colonies. *Proceedings of the First European Conference on Artificial Life*, Elsevier Publishing, Paris, 134–142.

[Cur04] Curran, D. & O'Riordan, C. (2004). Evolving Agents to play connect-four using cultural learning. *Proceedings of the 15th Irish Artificial Intelligence and Cognitive Science Conference* (AICS 2004).

[DeM04] DeMeyer, K. (2004). Foundations of Stochastic Diffusion Search [Väitöskirja]. The University of Reading, UK.

[DeM06] De Meyer, K., Nasuto, S. J. & Bishop, J. M., (2006), Stochastic Diffusion Optimisation: the application of partial function evaluation and stochastic recruitment in Swarm Intelligence optimisation, *Stigmergic Optimization*, vol. 31, 185–207.

[Dio07] Diosan, L. & Oltean, M. (2007). Observing the swarm behaviour during its evolutionary design. *Proceedings of the 2007 GECCO conference companion on Genetic and evolutionary computation.* 2667–2674.

[Dor04] Dorigo, M. & Stützle, T (2004) – *Ant Colony Optimization*. Massachusetts Institute of Technology, USA.

[Gre95] Grech-Cini, H. J. (1995). *Locating Facial Features.* [Väitöskirja]. University of Reading, UK.

[Hay99] Haykin, S. (1999). *Neural Networks – A comprehensive foundation, Second Edition*. Prentice-Hall, New Jersey.

[Her06] Hereford, J. M. (2006). A Distributed Particle Swarm Optimization Algorithm for Swarm Robotic Applications. *Evolutionary Computation, 2006. CEC 2006. IEEE Congress on*, 1678–1685.

[Hsi07] Hsieh, C.-T., Chen, C.-M. & Chen, Y.-P. (2007). Particle Swarm Guided Evolution Strategy. *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, 650–657.

[Jia07] Jiang, M., Luo, Y. & Yang, S. (2007). Stagnation Analysis in Particle Swarm Optimization. *Swarm Intelligence Symposium, 2007. SIS 2007. IEEE*, 92–99.

[Ken95] Kennedy, J. & Eberhart, R. (1995). Particle swarm optimization. *Neural Networks, 1995. Proceedings., IEEE International Conference on*, vol. 4, 1942–1948.

[Ken00] Kennedy, J. Stereotyping: improving particle swarm performance with cluster analysis. (2000). *Evolutionary Computation, 2000. Proceedings of the 2000 Congress on*, vol.2, 1507–1512.

[Ken01] Kennedy, J. & Eberhart, R.C. (2001). *Swarm Intelligence*. Morgan Kaufmann Academic Press, San Francisco.

[Ken05] Kennedy, J. (2005). Dynamic-Probabilistic Particle Swarms. *Proceedings of the 2005 conference on Genetic and evolutionary computation*, 201–207.

[Kle06] Kleinberg, J. & Tardos, É. (2006) *Algorithm Design*. Courier Westford, USA.

[Kod07] Koduru, P., Das, S. & Welch, S. M. (2007). Multi-Objective Hybrid PSO Using ε-Fuzzy Dominance. *Proceedings of the 9th annual conference on Genetic and evolutionary computation GECCO '07*, 853–860.

[Kro04] Krohling, R. A., Hoffman, F & Coelho, Ld.S (2004). Co-Evolutionary Particle Swarm Optimization for Min-Max Problems using Gaussian Distribution. *Evolutionary Computation, 2004. CEC2004. Congress on*, vol.1, 959–964.

[Li04] Li, X. (2004). Adaptively Choosing Neighbourhood Bests Using Species in a Particle Swarm Optimizer for Multimodal Function Optimization. *GECCO-2004*, K. Deb, et al., Eds. Springer-Verlag, 105–116.

[Li07] Li, X. (2007). A Multimodal Particle Swarm Optimizer Based on Fitness Euclidean-distance Ratio. *Proceedings of the 9th annual conference on Genetic and evolutionary computation,* 78–85.

[Lia06] Liang, J. J., Qin, A. K., Suganthan, P. N. & Baskar, S. (2006). Comprehensive Learning Particle Swarm Optimizer for Global Optimization of Multimodal Functions. *Evolutionary Computation, IEEE Transactions on*, vol. 10, no. 3, 281–295.

[Liu05] Liu, B-F., Chen, H-M., Chen, J-H., Hwang, S-F. & Ho S-Y. (2005). MeSwarm: Memetic Particle Swarm Optimization. *Proceedings of the 2005 conference on Genetic and evolutionary computation,* 267–268.

[Løv01] Løvbjerg, M., Rasmussen, T. K. & Krink, T. (2001). Hybrid Particle Swarm Optimiser with Breeding and Subpopulations. *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO-2001.*

[Men04] Mendes R., Kennedy J., Neves J. (2004). The Fully Informed Particle Swarm: Simpler, Maybe Better. *IEEE Trans. Evolutionary Computation* vol 8, no. 3, 204–210.

[Mes04] Messerschmidt, L. & Engelbrecht, A. P. (2004). Learning to Play Games Using a PSO-Based Competitive Learning Approach. *IEEE Transactions On Evolutionary Computation*, vol. 8, no. 3, 280–288.

[Mes06] Messerschmidt, L. (2006). Using Particle Swarm Optimization to Evolve Two-Player Game Agents [Pro gradu]. University of Pretoria, South Africa.

[Mic00] Michalewicz, Z. & Fogel, D.B. (2000). *How to Solve It: Modern Heuristics*. Springer-Verlag, Germany.

[Mon05] Monson, C. K. & Seppi, K. D. (2005). Bayesian Optimization Models for Particle Swarms. *Proceedings of the 2005 conference on Genetic and evolutionary computation*, 193–200.

[Mon06] Monson, C. K. & Seppi, K. D. (2006). Adaptive Diversity in PSO. *Proceedings of the 8th annual conference on Genetic and evolutionary computation*, 59–66.

[Nas99] Nasuto, S.J. & Bishop, J. M. (1999). Convergence Analysis of Stochastic Diffusion Search. *Journal of Parallel Algorithms and Applications*, vol. 14, 89–107.

[Nor03] Norvig, P. (2003) – *Artificial Intelligence: A Modern Approach, Second Edition*. Pearson Education, USA.

[O'Ne06] O'Neill, M. & Brabazon, A. (2006). Self-Organizing Swarm (SOSwarm): A Particle Swarm Algorithm for Unsupervised Learning. *Evolutionary Computation, 2006. CEC 2006. IEEE Congress on*, 634–639.

[Pas06] Pasupuleti, S. & Battiti R. (2006). The Gregarious Particle Swarm Optimizer (G-PSO). *Proceedings of the 8th annual conference on Genetic and evolutionary computation,*. 67–74.

[Pol05] Poli, R., Chio, C. D. & Langdon, W. B. (2005). Exploring Extended Particle Swarms: A Genetic Programming Approach. *Proceedings of the 2005 conference on Genetic and evolutionary computation*, 169–176.

[Pol07] Poli, R. (2007). On the Moments of the Sampling Distribution of Particle

Swarm Optimisers. *Proceedings of the 2007 GECCO conference companion on Genetic and evolutionary computation,* 2907–2914.

[Rat04] Ratnaweera, A., Halgamuge, S. K. & Watson, H. C. (2004). Self-Organizing Hierarchical Particle Swarm Optimizer With Time-Varying Acceleration Coefficients. *Evolutionary Computation, IEEE Transactions on*, vol. 8, no. 3, 240–255.

[Sch02] Schneider, M. O. & Garcia Rosa, J.L. (2002). *Proceedings of the VII Brazilian Symposium on Neural Networks (SBRN'02)*, 236–241.

[Sen97] Sen, S. & Arora, N. (1997). Learning to take risks. *AAAI-97 Workshop on Multiagent Learning*, 59–64.

[Set05] Settles, M. & Soule, T. (2005). Breeding Swarms: A GA/PSO Hybrid. *Proceedings of the 2005 conference on Genetic and evolutionary computation*, 185–192.

[Shi99] Shi, Y. & Eberhart, R. C. (1999). Empirical study of particle swarm optimization. *Proc. IEEE Int. Congr. Evolutionary Computation*, vol. 3, 101–106.

[Sil02] Silva, A., Neves, A. & Costa, E. (2002). An Empirical Comparison of Particle Swarm and Predator Prey Optimisation. *Proceedings of the 13th Irish International Conference on Artificial Intelligence and Cognitive Science*, 103–110.

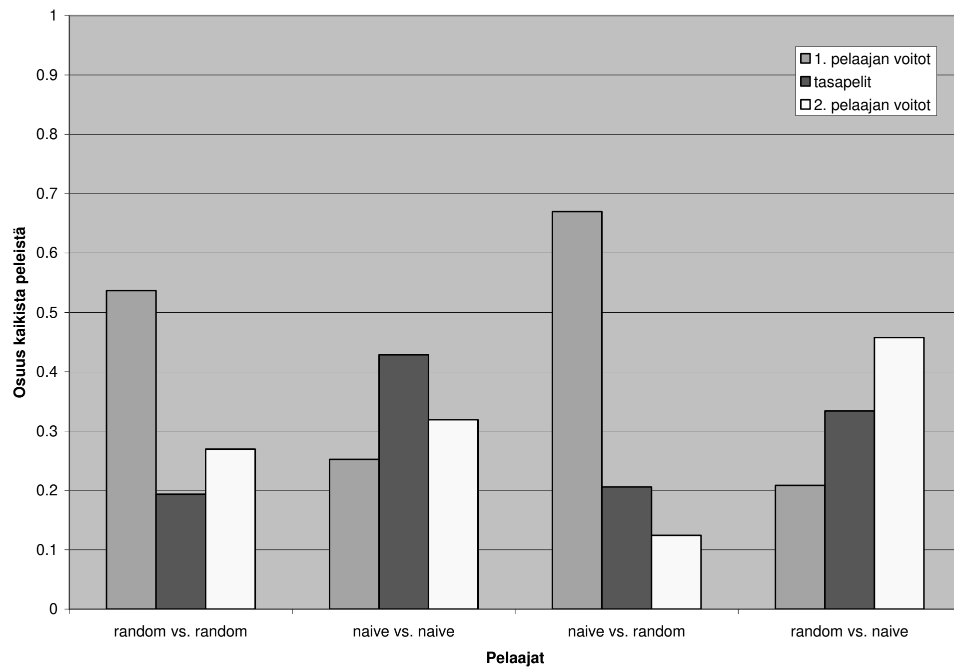
[Sut06] Sutton, A. M., Whitley, D., Lunacek, M. & Howe, A. (2006). PSO and Multi-Funnel Landscapes: How cooperation might limit exploration. *Proceedings of the 8th annual conference on Genetic and evolutionary computation*, 75–82.

[vdB02] van den Bergh, F. (2002). An Analysis of Particle Swarm Optimizers. [Väitöskirja]. Department of Computer Science, University of Pretoria, Pretoria, South Africa. http://www.cs.up.ac.za/cs/fvdbergh/publications/phd_thesis.ps.gz (tarkistettu: 13.11.2007).

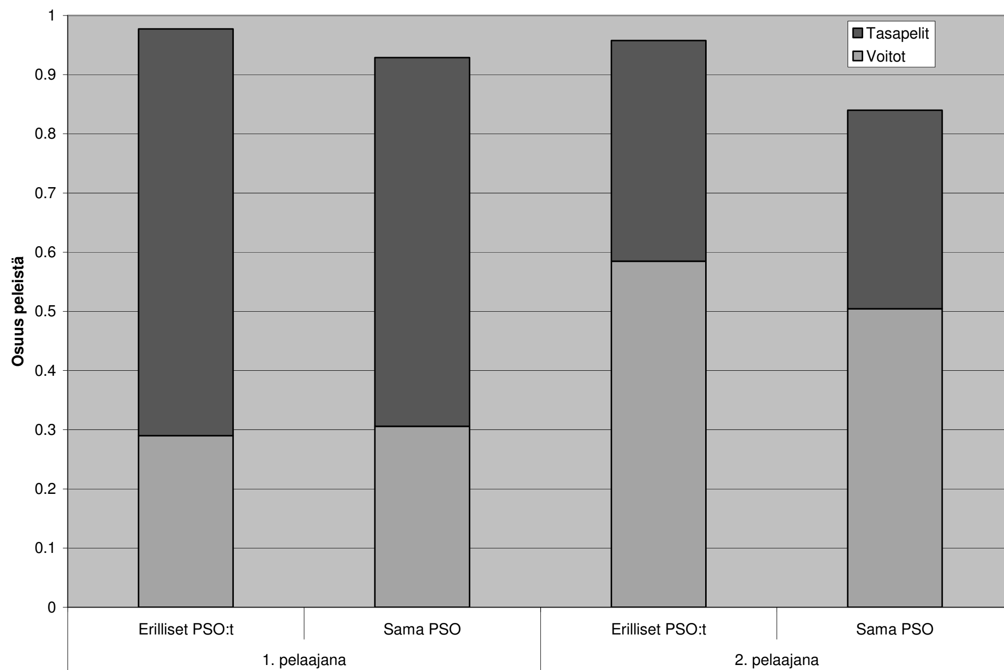
[Wol96] Wolpert, D. H. & MacReady, W. G. (1996). No Free Lunch Theorems for Optimization. *IEEE Transactions on Evolutionary Computation,* vol. 1, no. 1, 67–82.

[Zha03] Zhang, W.-J. & Xie, X.-F. (2003). DEPSO: Hybrid Particle Swarm with Differential Evolution Operator. *Systems, Man and Cybernetics, 2003. IEEE International Conference on*, vol.4, 3816–3821.

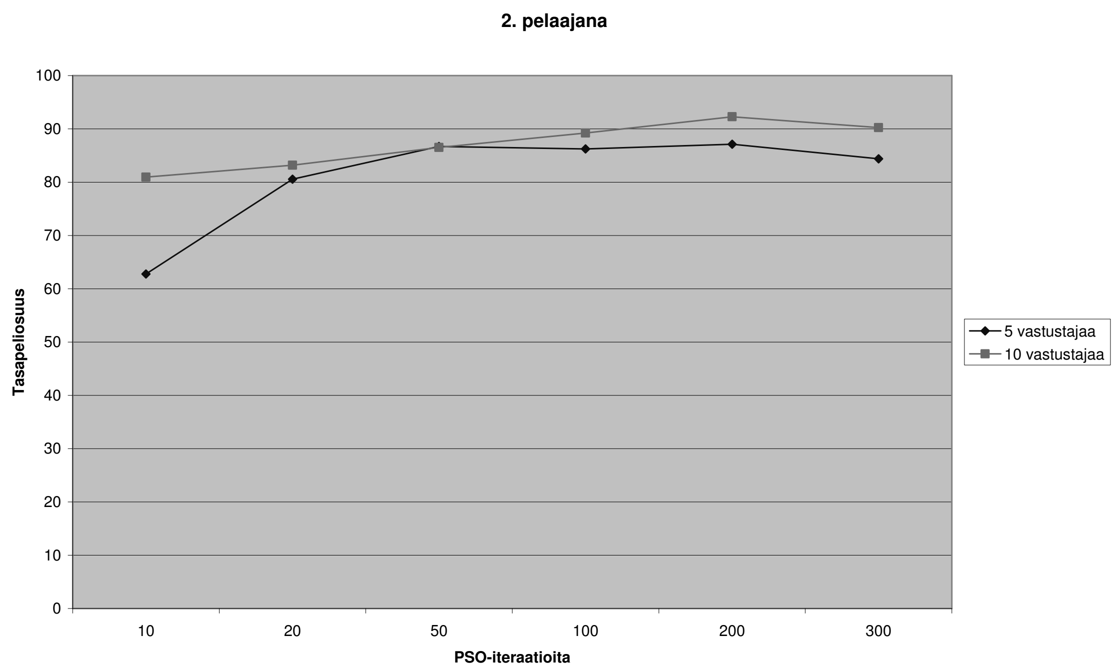
Liite 1. Kuvia saaduista tuloksista



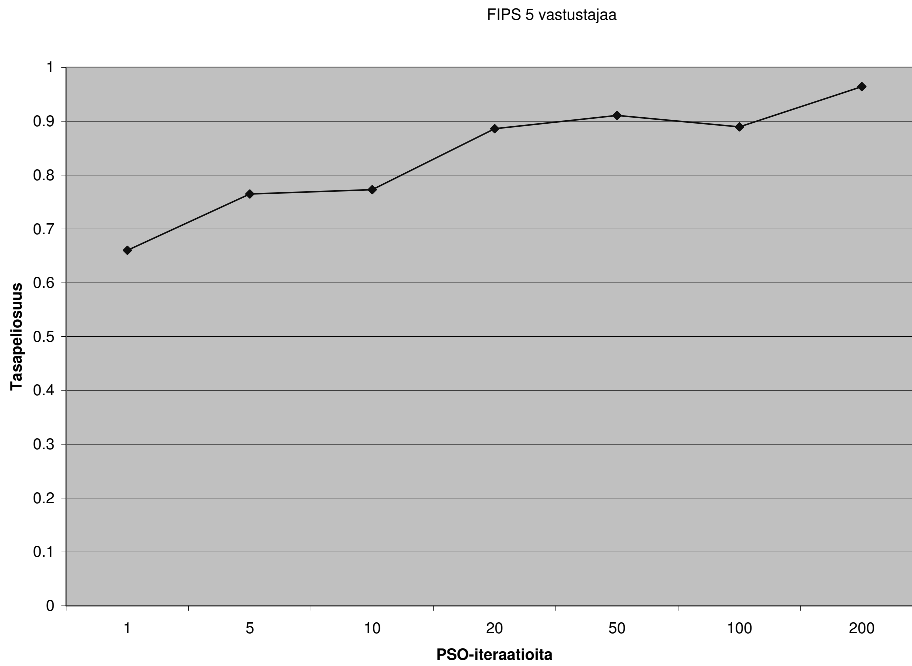
Kuva 14. Referenssipelaajien keskinäinen pelitaso 3x3-pelilaudalla. Satunnaisen pelaajan (random) ja Naiven välisten pelitulosten osuus (ykköspelaajan voitot, tasapelit, kakkospelaajan voitot).



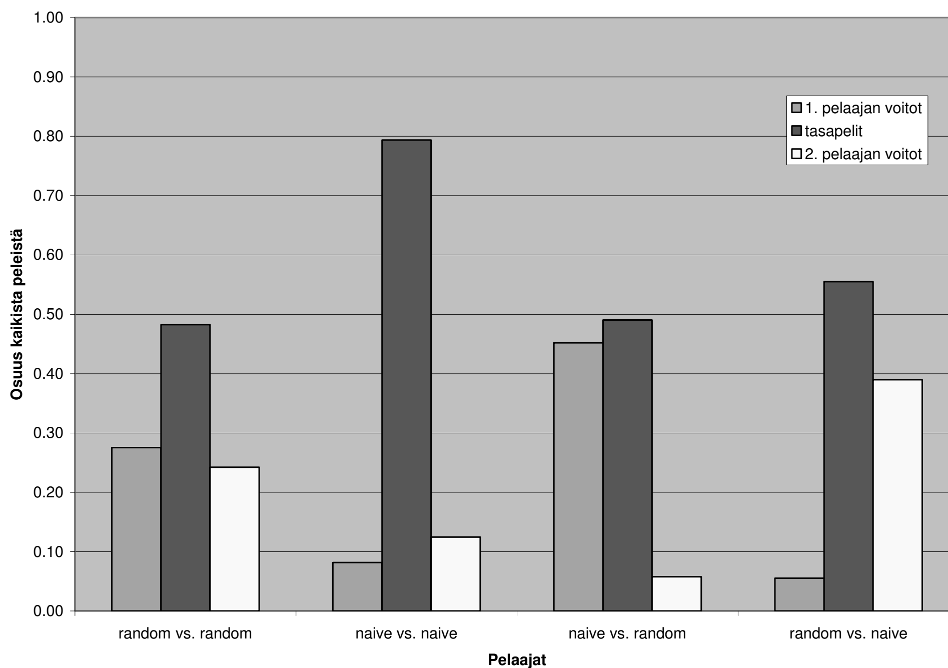
Kuva 15. FIPS:lla kehitetyn pelaajan voitto- ja tasapeliosuudet Naive-pelaajaa vastaan käydyissä peleissä sekä ykkös- että kakkospelaajana. Käytettiin kahta erilaista versiota: a) kahta pelaajaa, joista toinen kehitettiin ykkös- ja toinen kakkospelaajaksi. b) pelaajaa, joka toimi sekä ykkös- että kakkospelaajana (sama PSO)



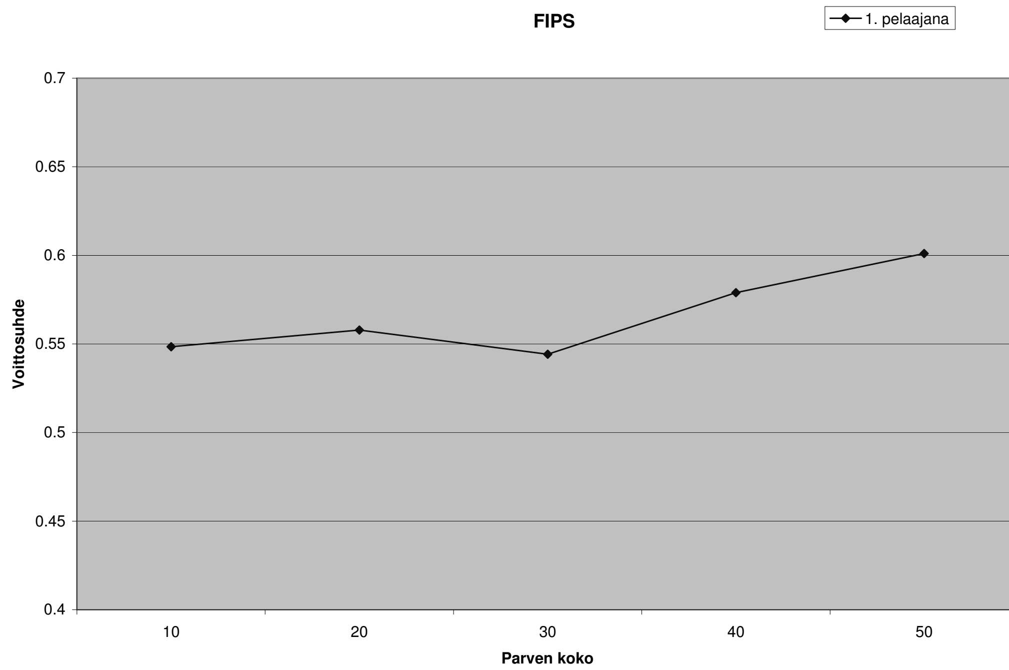
Kuva 16. Tasapelien osuus CLPSO:lla kehitetyn pelaajan pelatessa kakkospelaajana minimax-9:ää vastaan. Rinnakkaisevoluutiossa käytettäviä vastustajia on 5 ja 10.



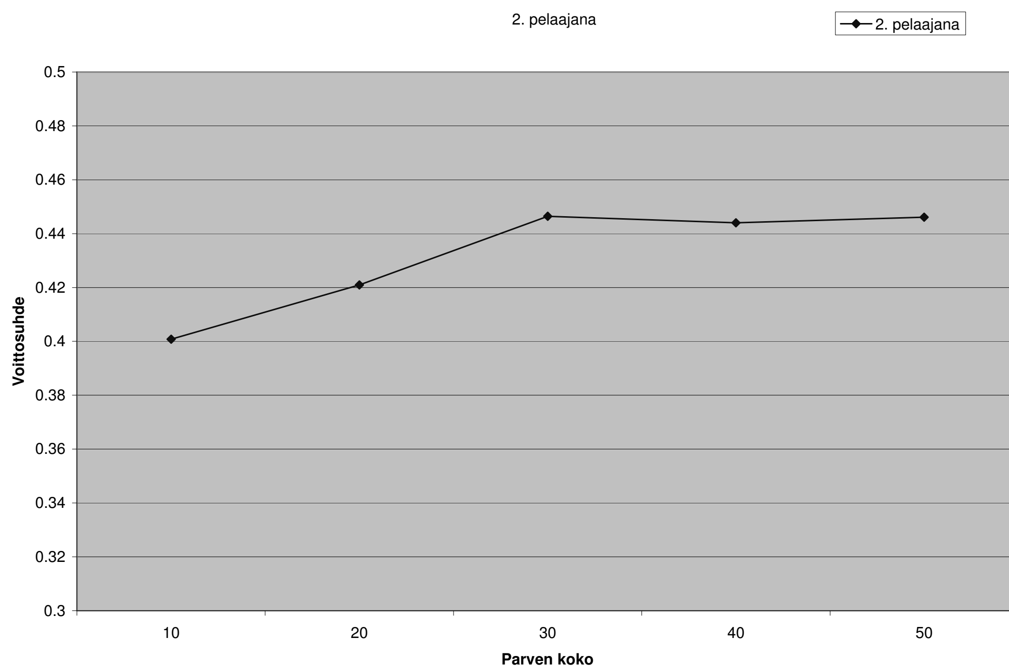
Kuva 17. FIPS:llä kehitettyjen pelaajien tasapeliosuus PSO-iteraatioiden funktiona pelattaessa kakkospelaajana minimax-9:ää vastaan.



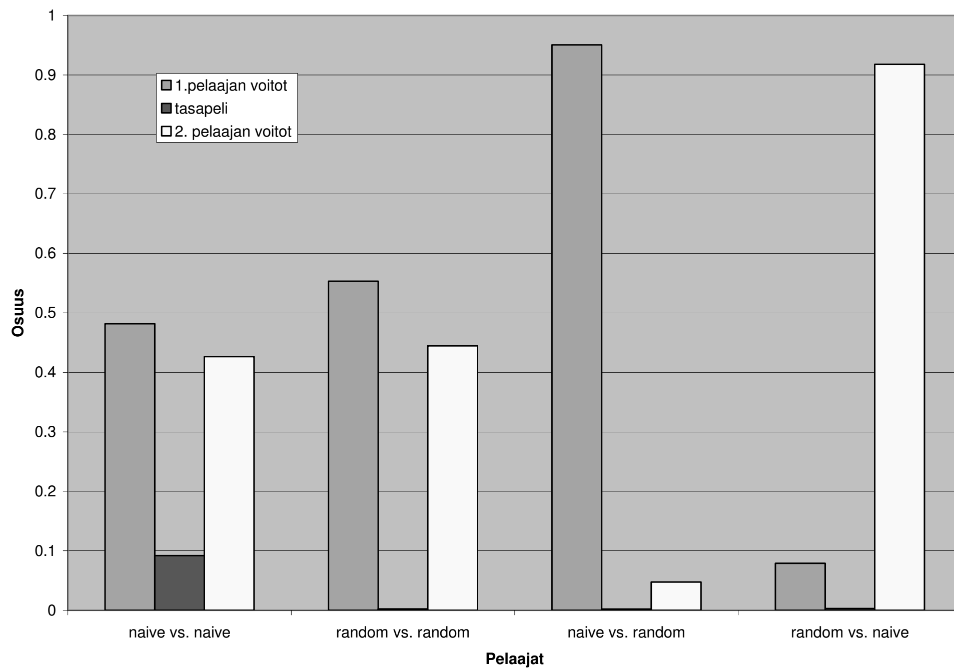
Kuva 18. Referenssipelaajien keskinäinen pelitaso 4x4-laudalla. Satunnais- ja Naive-pelaajan pelaamien pelien lopputulososuudet (ykköspelaajan voitot, tasapelit, kakkospelaajan voitot).



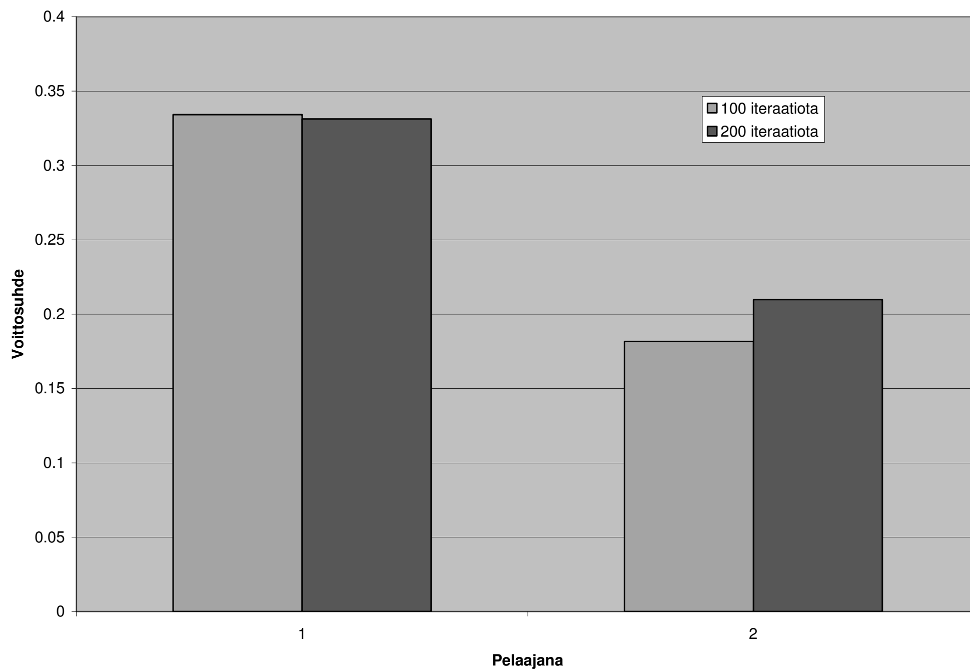
Kuva 19. FIPS:llä kehitetyn pelaajan voittosuhteet pelatessa ykköspelaajana satunnaispelaajaa vastaan parven koon funktiona.



Kuva 20. FIPS:llä kehitetyn pelaajan voittosuhteet pelatessa kakkospelaajana satunnaispelaajaa vastaan parven koon funktiona.



Kuva 21. Referenssipelaajien välinen pelitaso 7x6-laudalla. Pelien lopputulososuudet (ykköspelaajan voitot, tasapelit, kakkospelaajan voitot) Naiven ja satunnaispelaajan pelatessa ykkös- sekä kakkospelaajana.



Kuva 22. FIPS:in voittosuhde pelatessa sekä ykkös- että kakkospelaajana Naivea vastaan, kun iteraatioita on 100 ja 200.