Particle Swarm Optimization Applied to the Chess Game

João António Duro and José Valente de Oliveira

Abstract—To the best of the authors’ knowledge this paper investigates for the first time the applicability of particle swarm optimization (PSO) to a chess player agent endowing it with learning abilities, i.e. allowing the agent to improve its performance based on its experience.

A minimax algorithm with alpha beta pruning is used to select the next move of the chess agent. The performance of the agent strongly depends on the heuristic evaluation function available to the minimax algorithm. In this work, board features such as material strength, piece mobility, pawn structure, king safety and control of the centre are used in a parameterized board evaluation function whose weights are optimized using PSO.

The simulation results included, illustrate both the feasibility of the proposed approach and reveals that on average, PSO can provide faster learning results than simulated annealing under similar experimental conditions, especially in the presence of bounded computing time. Unfortunately, results also show that, for this application, PSO is highly sensitive to initial conditions.

I. INTRODUCTION

ONE of the artificial intelligence (AI) objectives is to capture human knowledge and expertise in programs that emulate the behaviour of the human experts. The so-called weak AI is essentially concerned with the design of software agents that simulate intelligent behaviour, rather than the mechanisms that underlie this behaviour.

Another known goal of AI is the creation of machines that can learn from their experience without significant human intervention [1].

For many years, the game of chess has served as a testing ground for efforts in artificial intelligence, both in terms of computers playing against other computers and computers playing against humans. Chess is appreciated for its “difficulty”, i.e., by its high branch-factor. The average number of possible movements that a player can choose from is 35. Recent progression in chess games is primarily due to hardware advances rather than to major breakthrough in search or optimization algorithms. One of the most famous chess computer games is IBM’s Deep Blue. Deep Blue was able to win one game against the world champion Gary Kasparov, in February of 1996. IBM maintains a page of the event on its website1. Its hardware was composed of 32 processors and able to evaluate 200 million alternative positions per second. By contrast, the computer that executed Belle, the first program to earn the title of U.S. master in 1983 was more than 100 000 times slower. Faster computing and optimization programming allows a chess program to evaluate chessboard position further into the prospective future. Such a program can then choose current moves that can be expected to lead to better outcomes which might not be seen by a program running on a slower computer or with inefficient programming.

Typically, chess agents rely on a database of opening moves and endgame positions, relying only in adversarial search for intermediate positions. To deal with the high branching factor, search algorithms employ a heuristic function for estimating the goodness of the many chess board configurations. This function usually combines features regarding the values assigned to individual pieces (material strength), piece mobility, pawn structure, king safety, control of the centre and others that are used to assign values to pieces based on their position (positional values) on the chess board. The parameters that weight these features are set by human experts but can be optimized using a machine learning approach. Furthermore, an optimization algorithm can be employed to discover features that lead to improved game playing.

In this paper we aim to investigate the merits, if any, of applying particle swarm for learning to estimate the weights of the heuristic evaluation function. For assessing the merits of the approach, PSO estimates are compared with Simulated Annealing (SA) results, under similar experimental conditions. The idea of applying SA to chess aims at avoiding heuristic optimization from stopping at a local minimum solution.

During the experiments a learning chess agent plays against another chess agent equipped with a fixed expert-tuned heuristic evaluation function.

Two sets of experiments were made between these two players. Each set corresponds to a single type of experiment. In the first type, the learning agent uses only a material strength based heuristic evaluation function while the fixed agent employs a set of features including king safety, pawn structure, control of the centre and lines of attack. With this type of experiment the learning agent aims to discover the relative importance of each chess figure. In other words, one aims at investigating whether our learning scheme is able to find out what is more relevant for achieving a victory: a pawn or a queen? The knight or the bishop? In the second type of experiments both players employ the same set of features in the evaluating function. This type of experiment aims at checking whether the learning player can outperform the fixed heuristic agent.

The paper is organized as follows. After this Introduction, Section II formulates our learning task as a parametric inductive learning task. Sections III and IV briefly revise PSO and Simulated Annealing, specifying the settings and options taken in this study. Section V presents the feature evaluators employed in the heuristic evaluating function. Section VI describes their experimental conditions and obtained results with respective analysis. Finally, Section VII ends the paper drawing the main conclusions and pointing out some future lines of work.
II. PROBLEM FORMULATION

We formulate our learning task as a parametric inductive learning problem.

We start by assuming that a function exists mapping any given chessboard $b$ to a numeric value indicating the goodness of the $b$ relatively to our objective of winning the game. Such function, $V(b)$, will be such that:

$$
V(b) =
\begin{cases}
100 & \text{if } b \text{ corresponds to our victory} \\
-100 & \text{if } b \text{ corresponds to a loss} \\
0 & \text{if } b \text{ stands for a draw}
\end{cases}
(1)
$$

While the values of $V(b)$ are easily computed when $b$ is a terminal state, i.e., when $V(b)$ measures end game results, there is no efficient method to compute $V(b)$ for an arbitrary $b$. For a non-terminal state $b$, $V(b) = V(\text{best'})$, where best' is the best state that can be reached departing from $b$ using a optimal playing strategy.

Interestingly and conveniently enough, it can be shown that an optimal estimate of $V(b)$, $\hat{V}(b)$, is given by [2]:

$$
\hat{V}(b) = \hat{V}(\text{Successor}(b))
(2)
$$

where $\hat{V}$ is the learner’s current approximation to $V$ and $\text{Successor}(b)$ denotes the next board state departing from $b$ for which it is again the learners turn to move.

Using PSO no special properties on the linearity or differentiability of the hypothesis $\hat{V}(b)$ are required. This is an advantage over more classical optimization/learning schemes. In this case, one can simply say that

$$
\hat{V}(b) = \Psi(\hat{F}(b), \Omega)
(3)
$$

where $\Psi$ is a possibly non-linear function, parameterized in $\Omega$, mapping the feature space into $\mathbb{R}$. Moreover, $\hat{F}(b) = [f_1(b), \ldots, f_n(b)]^T$ are $n$ features characterizing the chess board $b$, $\Omega$ being defined as $\Omega = [\omega_0, \ldots, \omega_n]^T$, i.e., as a vector of $n+1$ real-valued parameters.

The learning task consists in the minimization of $E$, the sum of the squared error between $\hat{V}(b)$ and the values predicted by the hypothesis $V(b)$, for all $b$. More specifically:

$$
E \equiv \sum_{(b, \hat{V}(b)) \in \text{training examples}} (\hat{V}(b) - V(b))^2
(4)
$$

Since our goal is to investigate the applicability and performance of PSO as a tool for developing chess agent learners, the simplest hypothesis for $V(b)$ is adopted, i.e.

$$
\hat{V}(b) = \hat{F}^T(b)\Omega
(5)
$$

$$
\hat{V}(b) = \omega_0 + \omega_1 f_1(b) + \omega_2 f_2(b) + \ldots + \omega_n f_n(b)
$$

III. THE PARTICLE SWARM OPTIMIZER

PSO is a general-purpose optimization technique developed by Eberhart and Kennedy [3]. This technique was inspired by the concept of swarms in nature, such as bird flocking, fish schooling or insect swarming. The idea is that individual members of the swarm can profit from the discoveries and previous experiences of all other members of the swarm during the search for the optimum solution.

In the algorithm each individual in the particle swarm (hereafter referred to as a particle) is represented as an $n$-dimensional vector $\vec{\omega}$, for which we seek some kind of optimum.

A neighbourhood is defined on the population as a mapping from each particle to some subset of the population. Several different types of neighbourhood can be employed. Fig. 1 present three of the more popular ones.

Based on some experimental evidence reporting the superiority of the Von Neumann topology [4], we adopt this swarm neighbourhood topology here.

A population of size 10 is used with a Von Neumann topology, which means each particle has 4 neighbours. Two particles are said to be neighbours if they are exactly one edge away from each other.

A velocity vector $\vec{v}$, which defines a particle’s current motion through the weight-space, a vector $\vec{p}$ that defines the best position achieved by the particle and a vector $\vec{b}$ containing the best solution vector seen by the neighbourhood so far, are tracked for each particle. Furthermore, a fitness value for $\vec{v}$, $\vec{p}$ and $\vec{b}$ are stored for each particle.

Preceding each and every iteration of the PSO algorithm is an evaluation phase, during which the fitness $fit$ of the current weight vector $\vec{\omega}$ is determined. This is achieved by updating the personal $\vec{p}$ and neighbourhood $\vec{b}$ best positions and accumulating the particle’s fitness.

The above vectors are updated according to the following equation:

$$
\vec{v}_t = \varphi \cdot \vec{v}_{t-1} + \phi_1 (\vec{p} - \vec{\omega}_{t-1}) + \phi_2 (\vec{b} - \vec{\omega}_{t-1})
\vec{\omega}_t = \vec{\omega}_{t-1} + \vec{v}_t
(6)
$$

The momentum, $\varphi$, can be used to control how "light" particles are, i.e., how easy it is to accelerate them. The parameters $\varphi$, $\phi_1$ and $\phi_2$ define the kinetic behaviour of particles. The pseudo-code of the algorithm can be seen in Fig. 2.

As usual, the particles of the swarm are initialized with random positions $\vec{\omega}_i$ and velocity $\vec{v}_i$. The fitness function $fit$ is evaluated, using the particle’s positional coordinates as input values. The position and velocity for each particle are adjusted and the function $fit$ is evaluated with new coordinates at each time step. The variables $\phi_1$ and $\phi_2$ are random positive numbers drawn from a uniform distribution.

The term $v_{ad}$ is limited to the range $\pm V_{max}$, in order to control velocity explosions [5]. The values of the elements in $b_i$ are determined by comparing the best performance of all the members in the neighbourhood. Thus, $b_i$ represents the best position found by any member of the neighbourhood. The $N_{max}$ stands for a pre-defined maximum number of iterations allowed for the algorithm. Once it achieves $0$ it
Algorithm III.1: PSO($V_{\text{max}}, \varphi, c_1, c_2, N_{\text{max}}$)

Initialize population
repeat
\[ i \leftarrow 1 \text{ to Population Size} \]
\[ \text{if } f_{i}(\bar{x}_i) < f_{i}(\bar{p}_i) \]
\[ \text{then } \bar{p}_i = \bar{x}_i \]
\[ \bar{b}_i = \text{minimum(}p_{\text{neighborhood}(j)}\text{)} \]
\[ \text{for } d = 1 \text{ to Dimension} \]
\[ \phi_1 = c_1 \cdot U \sim (1, -1) \]
\[ \phi_2 = c_2 \cdot U \sim (1, -1) \]
\[ v_{id} = \varphi \cdot v_{id} + \phi_1(p_{id} - \omega_{id}) + \phi_2(b_{id} - \omega_{id}) \]
\[ v_{id} = \text{sign}(v_{id}) \cdot \text{min}(\text{abs}(v_{id}, V_{\text{max}})) \]
\[ \omega_{id} = \omega_{id} + v_{id} \]
until termination criterion is met or $N_{\text{max}}$

Fig. 2. Pseudo-code of the PSO algorithm.

stops the algorithm iterations before the desired convergence has been achieved. The function $fit$ used in this work is the equation 4 referenced in Section II.

The parameter values chosen are very important for the desired convergence and stability of the PSO and they are also important for the understanding of the results obtained. The acceleration constants, $c_1$ and $c_2$, have been restricted considering a work developed by Kennedy [6] that prevents oscillation of particles, by applying the next equation:

\[ c_1 + c_2 \leq 4 \]

A comparatively larger $c_1$ value will steer the particle in the direction of its previous best solution, while a larger $c_2$ value will steer it toward the neighbouring best solution. In the work developed by Cornelis Franken [7] where PSO was applied to a checkers game, the acceleration constants’ values were, as in this work, chosen as $c_1 = c_2 = 1$.

The inertia weight, $\varphi$, has a significant influence on convergence by adjusting the retained size of the previous velocity for the current time step. A large inertia weight value facilitates global search, while a smaller inertia weight facilitates local search. Van den Bergh [8] derived the following equation that limits the parameter values to achieve a local solution:

\[ \varphi > \frac{(c_1 + c_2)}{2} - 1 \]

In this work $\varphi = 0.8$ is used for the inertia weight. There is some experimental evidence that this value allows for better convergence swarms, cf. [5].

In order to avoid particle velocity explosions in the swarm that can cause lower performance and no convergence behaviour, a $V_{\text{max}}$ of value 1 was used.

The parameters presented above have been used in the first set of experiments, in the second set of experiments other parameters have been chosen in order to allow the algorithm to take more time while the particles converge.

IV. SIMULATED ANNEALING


The name and inspiration come from annealing in metallurgy, a technique involving heating and controlled cooling of a material to increase the size of its crystals and reduce their defects [10]. The heat causes the atoms to become unstuck from their initial positions (a local minimum of the internal energy) and wander randomly through states of higher energy the slow cooling gives them more chances of finding configurations with lower internal energy than the initial one.

By analogy with this physical process, each step of the SA algorithm replaces the current solution by a random “nearby” solution, chosen with a probability that depends on the difference between the corresponding function values and on a global parameter $T$ (called the temperature), that is gradually decreased during the process. The dependency is such that the current solution changes almost randomly when $T$ is large, but increasingly “downhill” as $T$ goes to zero. The allowance for “uphill” moves saves the method from becoming stuck at local minima - which are the bane of greedier methods [11].

The SA algorithm used in this work is given in Fig. 3. The idea in applying this algorithm to chess is to prevent the algorithm from stopping at the local minima solution and to be able to compare it with the PSO approach. Knowing that SA has this specific particularity, it is not known, prior to this work, if the application to the chess game playing environment will succeed.

In our work, the SA algorithm initially ran 1024 interactions, which is the number of interactions necessary for the temperature heating that accepts non-favourable movements with a probability of 50% and favourable movements with a 100% probability. The algorithm calculates the variance of the movements and then multiplies them by two in order to obtain the final temperature.

There is a mutation function applied in order to expand the search area for the algorithm to be able to find other solutions other than the local ones.

V. EVALUATING BOARD FEATURES

At the leaves of the minimax tree, board positions are assessed using several feature evaluators, viz. functions which reveal the merits of the board.

The employed features are based on the GNU Chess implementation and are as follows [13]:

- $V_1$: Material Balance: This feature counts the material presented in the board. The queen ($f_1$) worth 980, rook ($f_2$) 550, bishop ($f_3$) 330, knight ($f_4$) 330 and pawn ($f_5$) 100. The evaluation function has the form $[(f_{1p} \times 980) - (f_{1o} \times 980)] + [(f_{2p} \times 520) - (f_{2o} \times 520)] + [(f_{3p} \times 330) - (f_{3o} \times 330)] + [(f_{4p} \times 330) - (f_{4o} \times 330)] + [(f_{5p} \times 100) - (f_{5o} \times 100)]$. In the equation $p$ is the current player and $o$ the opponent.

- $V_2$: Pawn Structure. Here only two types of pawns are considered, doubled or tripled pawns ($f_2$) and isolated pawns ($f_2$). The evaluation function has the form $(-f_1 \times 12) + (-f_2 \times 1)$. 

2008 IEEE Congress on Evolutionary Computation (CEC 2008) 3705
Algorithm IV.1: SA($\bar{p}$, $N_{\text{max}}$)

test, $\bar{c}$ ← Clone($\bar{p}$)
init ← double[1024]
$tp, et, cs, ts, dec ← 0.0$
bs, mi ← maximum double

Initial temperature computation

for $i ← 1$ to 1024

\[
\begin{align*}
\text{test} & ← \text{Copy}(\bar{c}) \\
\text{Mutate(test)} & \\
\text{init} & ← \text{Score(test)} \\
\text{if} \ (Abs(\text{init}) < cs) & \text{\textbf{then}} \ (\text{mi} ← \text{init} - cs) \\
\text{if} \ (\text{init} < cs \text{ or } \text{Rand} \text{ (true , false)}) & \text{\textbf{then}} \ (\bar{c} ← \text{Copy(test)})
\end{align*}
\]

$cs ← \text{init}$
$tp ← \text{Stdec(init)} \times 2.0$
$et ← \frac{mi}{2.0}$
$dec ← \text{Decrement}(tp, et, N_{\text{max}})$

return $(bs)$

beginning of the temperature cooling

for $i ← 1$ to $N_{\text{max}}$

\[
\begin{align*}
\text{test} & ← \text{Copy}(\bar{c}) \\
\text{Mutate(test)} & \\
\text{ts} & ← \text{Score(test)} \\
\text{if} \ (ts \neq cs \text{ and } (Abs(ts - cs)) < mi) & \text{\textbf{then}} \ (mi ← Abs(ts - cs)) \\
\text{et} & ← \frac{mi}{2.0}$
\]

\[
\begin{align*}
\text{if} \ (\text{Metrop}(cs, ts, tp)) & \text{\textbf{then}}
\begin{align*}
\bar{c} & ← \text{Copy(test)} \\
\text{cs} & ← ts
\end{align*}
\]

\[
\begin{align*}
\text{if} \ (bs > cs) & \text{\textbf{then}} \ (bs ← cs) \\
\text{if} \ (tp < 1) & \text{\textbf{then}} \ (tp ← tp \times dec)
\end{align*}
\]

\[
\text{tp} ← \text{tp} \times dec
\]

\[
\text{return} \ (bs)
\]

\[
\text{Fig. 3. Pseudo-code of the Simulated annealing algorithm [12].}
\]

- $V_3$ : King Safety. Here the features return 0 (false) or 1 (true). Is the king being attacked? ($f_1$), is the king in the centre of the board during the opening game? ($f_2$), is the king in an open file during the opening game? ($f_3$), is the king in a half open file during the opening game? ($f_4$), is there any open file adjacent to the king during the opening game? ($f_5$), is there any half open file adjacent to the king during the opening game? ($f_6$), is the king in the centre of the board during the end of the game? ($f_7$), is there any pawn of the same colour adjacent to the king? ($f_8$). The evaluation function has the form $(-f_1 \times 100) + (-f_2 \times 24) + (-f_3 \times 23) + (-f_4 \times 18) + (-f_5 \times 23) + (-f_6 \times 18) + (f_7 \times 36) + (f_8 \times 8)$.

- $V_5$ : Attack Lines. This feature counts the number of positions on the board that each piece is able to control ($f_1$). So after counting all the piece’s controlled positions the evaluation function has the form $f_1 \times 1$.

The global evaluation function used is, due to the reasons presented previously, a linear combination of the five features presented above, i.e. $V = \omega_0 + \omega_1 V_1 + \omega_2 V_2 + \omega_3 V_3 + \omega_4 V_4 + \omega_5 V_5$.

The static chess player, which serves as the adversary of our learning chess agents, uses $\omega_0 = 0$ while all other weights are equal to 1.

In a first set of experiments, the learning chess agent uses an evaluation function based only on the material balance. The weights associated to each chess figure are randomly initialized and learnt during the successive matches. The goal is to find out whether chess learners are able to discover plausible weights for each one of the chess pieces. There is a weight associated with each $p$ and $o$ from the Material Balance $V_1$ equation.

In a second type of experiment the learning chess agent employs all five features mentioned above. The goal being to verify whether the learning agent is able i) to improve its performance, and ii) to beat its adversary.

VI. EXPERIMENTAL CONDITIONS AND RESULTS

In this section, the conditions under which the experiments were conducted are presented. Some experimental results are given and a brief analysis of the results is provided.

A. Discovering the relative weight of the material

For the PSO algorithm the parameters chosen for the first type of experiments are presented in Table I. The number of particles chosen was intentionally low (10).

Each simulation performed 2000 games. In order to perform this amount of games PSO ran 200 iterations for each particle. The other parameters employed are given in Table I.

For keeping experiments comparable in computational terms, Simulated Annealing also performed 2000 games.

<table>
<thead>
<tr>
<th>Parameter name</th>
<th>Symbol</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum velocity</td>
<td>$V_{\text{max}}$</td>
<td>1</td>
</tr>
<tr>
<td>Inertia Weight</td>
<td>$\varphi$</td>
<td>0.8</td>
</tr>
<tr>
<td>Acceleration constant (Cognition)</td>
<td>$c_1$</td>
<td>1</td>
</tr>
<tr>
<td>Acceleration constant (Social)</td>
<td>$c_2$</td>
<td>1</td>
</tr>
<tr>
<td>Number of particles</td>
<td>$p$</td>
<td>10</td>
</tr>
<tr>
<td>Maximum number of iterations</td>
<td>$N_{\text{max}}$</td>
<td>200</td>
</tr>
<tr>
<td>Initial Weights</td>
<td>$\omega_p$</td>
<td>$U \sim (10, -10)$</td>
</tr>
</tbody>
</table>

TABLE I
PARAMETERS USED BY THE PSO BASED AGENT FOR THE FIRST TYPE OF EXPERIMENTS.

The result obtained using PSO can be seen in Fig. 4. It is possible to see that after 60 interactions most of the particles stabilize their weights and the algorithm has not performed a thorough search in the parameter space. It can be seen that it has only performed a search for parameter values between (8) and (−4).

Table II shows the final weights that the particles have obtained at the end of the simulations. From the PSO column, it is possible to see that despite the fast (premature?) convergence, the king has been the piece with most importance
(weight), queen is the second, third is the bishop, fourth is the knight, fifth the rook and last one is the pawn. The algorithm has been able to approximate the relative importance of most chess pieces after playing some games.

For the simulated annealing approach, the results obtained on Fig. 5 show that the algorithm was still doing a search in the parameter space but at the end of the experiment it was still far from plausible weights. It is evident than simulated annealing would benefit from extra computing time to obtain better results.

In Table II, the ω’s represent the weights of the global evaluation function. Each weight is combined with the feature present in the description according to equation 5.

If more time was available better feature evaluation would be achieved.

In this set of experiments, both PSO and SA have lost most of their games against the fixed player. This is not surprising, since the learning agents used a much simpler heuristic evaluation function when compared to the fixed player. We remember that the fixed player combines in his heuristic evaluation function all the features described in Section V.

B. Beating the adversary

In the second set of experiments, learning agents are equipped with all five available features. With this type of experiment the objective is to verify whether the learning agent is able to improve it’s performance and to beat its adversary.

The parameters chosen for the second experiment for the PSO based chess agent are presented in Table III.

Relative to the experiment in the previous section, Table III shows that most of the parameters have been reduced one order of magnitude, the intention is to prevent premature convergence.

Regarding the Simulated Annealing algorithm, since the algorithm needs to perform 1024 iterations in the beginning (which means 1024 games), as necessary for the temperature heating, it will only be able to play 976 games more (as 2000 games is for comparison purposes a pre-defined limited).

For the PSO based chess agent, weight evolution is presented in Fig. 6. It can be seen now that the particles have converged at a slower rate.

Fig. 7 shows the evolution of the weights during the SA algorithm application. The weights have spread in the search space faster than the PSO agent.

In Table IV it is possible to see the final values of the weights returned by the PSO and SA algorithm. For the PSO case the material balance feature has been the one that has the most credit. The pawn structure has fallen, the king safety and the number of attack lines have achieved a balance.

<table>
<thead>
<tr>
<th>Weight</th>
<th>PSO</th>
<th>SA</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ω_0</td>
<td>2.5014</td>
<td>−9.7328</td>
<td>no feature associated</td>
</tr>
<tr>
<td>ω_1</td>
<td>−1.3652</td>
<td>1.0566</td>
<td>white pawns</td>
</tr>
<tr>
<td>ω_2</td>
<td>0.094367</td>
<td>−7.3716</td>
<td>black pawns</td>
</tr>
<tr>
<td>ω_3</td>
<td>0.42542</td>
<td>29.593</td>
<td>black knights</td>
</tr>
<tr>
<td>ω_4</td>
<td>−2.8789</td>
<td>20.066</td>
<td>white knights</td>
</tr>
<tr>
<td>ω_5</td>
<td>1.3107</td>
<td>−1.8008</td>
<td>black bishops</td>
</tr>
<tr>
<td>ω_6</td>
<td>3</td>
<td>−17.333</td>
<td>white bishops</td>
</tr>
<tr>
<td>ω_7</td>
<td>2.0289</td>
<td>0.64604</td>
<td>black rooks</td>
</tr>
<tr>
<td>ω_8</td>
<td>−0.77434</td>
<td>−6.1234</td>
<td>white rooks</td>
</tr>
<tr>
<td>ω_9</td>
<td>1.7604</td>
<td>40.626</td>
<td>black queens</td>
</tr>
<tr>
<td>ω_10</td>
<td>−2.9027</td>
<td>−20.242</td>
<td>white queens</td>
</tr>
<tr>
<td>ω_11</td>
<td>−1</td>
<td>1</td>
<td>black king</td>
</tr>
<tr>
<td>ω_12</td>
<td>4</td>
<td>−20.263</td>
<td>white king</td>
</tr>
</tbody>
</table>

Table II: Weights obtained from PSO and SA based agent under adverse initial conditions.

Table III: Parameters used by the PSO based agent for the second type of experiments.
between them. For the SA case, more importance has been given to the pawn structure feature and all the others have been credited negative with the exception of $\omega_0$.

These values show again that the simulated annealing approach is not getting near to the desired weights, since these weights have not optimized the performance of the player affected by them. Any set of weights that affects the player performance considering the amount of games won would be considered the most desirable weights. One possible explanation is the mutation function used by the simulated annealing approach. This function is expanding the search area more widely than the PSO algorithm.

Considering the final results of games played during the optimization tasks, Fig. 8 shows in 5 bars the scores of the games for the PSO based agent. In the first 250 games both players had some equilibrium in terms of victories and losses. Afterwards, the number of victories for the learning agent has increased significantly. During the last games, when the learning agent stabilized weights, a reduction in the number of draws was observed.

![Particle Swarm Optimization](image1)

**Fig. 6.** Weight evolution for the PSO based agent under adverse initial conditions.

![Simulated Annealing](image2)

**Fig. 7.** Weight evolution for the SA based agent under adverse initial conditions.

**Table IV**

<table>
<thead>
<tr>
<th>Weight</th>
<th>PSO</th>
<th>SA</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\omega_0$</td>
<td>6.25489</td>
<td>6.96310</td>
<td>no feature associated</td>
</tr>
<tr>
<td>$\omega_1$</td>
<td>4.33272</td>
<td>-0.62538</td>
<td>material balance</td>
</tr>
<tr>
<td>$\omega_2$</td>
<td>-3.5483</td>
<td>23.5508</td>
<td>pawn structure</td>
</tr>
<tr>
<td>$\omega_3$</td>
<td>2.1201</td>
<td>-11.3953</td>
<td>king safety</td>
</tr>
<tr>
<td>$\omega_4$</td>
<td>2.2053</td>
<td>-14.8918</td>
<td>attack lines</td>
</tr>
<tr>
<td>$\omega_5$</td>
<td>-0.048237</td>
<td>-12.4767</td>
<td>control of the center</td>
</tr>
</tbody>
</table>

**Table V**

<table>
<thead>
<tr>
<th>Weight</th>
<th>PSO</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\omega_0$</td>
<td>6.25489</td>
<td>no feature associated</td>
</tr>
<tr>
<td>$\omega_1$</td>
<td>4.33272</td>
<td>material balance</td>
</tr>
<tr>
<td>$\omega_2$</td>
<td>-3.5483</td>
<td>pawn structure</td>
</tr>
<tr>
<td>$\omega_3$</td>
<td>2.1201</td>
<td>king safety</td>
</tr>
<tr>
<td>$\omega_4$</td>
<td>2.2053</td>
<td>attack lines</td>
</tr>
<tr>
<td>$\omega_5$</td>
<td>-0.048237</td>
<td>control of the center</td>
</tr>
</tbody>
</table>

**Fig. 8.** Evolution of game results for the PSO based agent under adverse initial conditions.

Fig. 9 shows the result of the games performed by the simulated annealing based agent. It is possible to see that no optimization has been performed and that, quite the opposite, a deterioration of the agent performance was observed. Clearly, the SA algorithm did not have enough time to optimize the weights.

In the following, we report the results of another similar experiment where only the initialization values of the particles were different. The particle’s final weights can be seen in the Table V. The “material balance” is one of the most important features but in this case the algorithm did not recognize this. Moreover, no significant distinction is observed for the remaining weights.

![Particle Swarm Optimization](image3)

**Fig. 10.** Evolution of the weights in this experiment. Since the beginning of the simulation the feature “material balance”, $\omega_1$, has increased slowly. According to Fig. 11 it is possible to see that since the beginning,
the learning player started to lose games. The particles did not get close a combination of winning weights. One explanation for this is the sensitivity of the PSO algorithm to the initial conditions. The initial weights have changed in every experiment according to the following distribution $U \sim (10, -10)$ as stated before.

Even for the simplest hypothesis used, viz. linear combination of features, the sensitivity to the initial conditions is not neglected and may lead to catastrophic results, such as those presented above. This effect can be even worse in the presence of non-linear heuristic evaluation functions, such as those based on artificial neural networks. A possible solution to this effect which obviates this sensitivity is to initiate particle positions with the best known weights of features.

In general, the obtained simulation results are promising and suggest that further research efforts are worth devoting to this idea for addressing the identified difficulties.

When properly optimized PSO has been able to quickly identify the relative importance of chess pieces. Also, it improved the board evaluation function weights and has been able to overcome a static expert-tuned opponent. This static opponent a standard set of heuristics as available from the GNU Chess.

The simulation results also show also that under the same computing restrictions the PSO based agent can be a faster learner than a Simulated Annealing based agent.

An identified issue in the application of PSO to the chess game is the high sensitivity of PSO to initial conditions, a problem that can be addressed by initializing particle positions with the best known feature’s weight, which means using the weights initially taken from the static player.

VII. Conclusions

To the best of the authors’ knowledge, the PSO algorithm has been applied for the first time to the chess player agent endowing it with learning capabilities.

REFERENCES


2008 IEEE Congress on Evolutionary Computation (CEC 2008) 3709

