An Evolutionary Algorithm coupled with the Hooke-Jeeves Algorithm for Tuning a Chess Evaluation Function

Eduardo Vázquez-Fernández
CINVESTAV-IPN
(Evolutionary Computation Group)
Departamento de Computación
Av. IPN No. 2508
Col. San Pedro Zacatenco
México D.F. 07300, MÉXICO
eduardovf@hotmail.com

Carlos A. Coello Coello
CINVESTAV-IPN
(Evolutionary Computation Group)
and UMI LAFMIA 3175
CNRS at CINVESTAV-IPN
Departamento de Computación
Av. IPN No. 2508
Col. San Pedro Zacatenco
México D.F. 07300, MÉXICO
ccoello@cs.cinvestav.mx

Feliú D. Sagols Troncoso
CINVESTAV-IPN
Departamento de Matemáticas
Av. IPN No. 2508
Col. San Pedro Zacatenco
México, D.F. 07360, MÉXICO
fsagols@math.cinvestav.edu.mx

***Abstract*—In a previous paper presented at CEC'2011, we reported the implementation of a chess engine based on evolutionary programming with a selection mechanism that relied on grandmaster's chess games. The objective was to decide the virtual players that would pass to the following generation. Here, we use these same techniques to adjust a larger number of weights (29 in this work instead of the 5 used in the previous one). The aim was to improve the rating of our chess engine. We also introduce here the use of a local search scheme based on the Hooke-Jeeves algorithm, which is adopted to adjust the weights of the best virtual player obtained in the evolutionary process. As our results indicate, this produced a further improvement in the rating of our chess engine. As in our previous work, the material values of the additional pieces considered here are similar to the values known from chess theory.**

I. Introduction

Chess is a game of perfect information in which there is no hidden information for the players. This means that, at a given position on the board, each player knows all of the possible movements. If one considers all possible movements, chess is an intractable game, because of its considerably large search space. Claude Shannon was the first to estimate that the total number of possible chess games is 10^{120}.

Because of its complexity, and the human interest that chess has attracted during many years, this game has been used, since the 1950s, as a benchmark to test a variety of artificial intelligence techniques.

In the 1950s, Alan Turing [19] designed a pioneering chess-playing program at a time at which getting access to digital computers was almost impossible. At about the same time, Claude Shannon [18] proposed two strategies to implement chess engines. The first used a "brute force" approach by performing an exhaustive search of the possible positions. This work was based on the application of the minimax algorithm on a game-tree. The second strategy proposed by Shannon adopted "artificial intelligence" to emulate the way in which humans play chess. Today, practically all chess engines use one of these two strategies or even combinations of them.

In 1958, Alex Bernstein wrote a chess-playing program that was able to play full games. However, this game was not very powerful and could be defeated by novice players.

In 1967, the MacHack VI program developed for the PDP-6 computer (from DEC) played, for the first time, against a human in a chess tournament. This program was able to achieve a rating of 1400 points.

In the $1970s$, chess tournaments officially began. At the end of this decade, the top chess programs available could play at the level of a human chess expert (around 2000 rating points).

The programs CHESS and Belle dominated the first two decades of competitions in the two major computer chess tournaments: the annual *North American Computer Chess Championships* (NACCC), and the *World Computer Chess Championships* (WCCC), which was held every three years.

It is worth indicating that, in the $1970s$, the main chess programs introduced the use of hash tables which allowed the storage of information about positions that had already been searched. This way, if the same position was reached again, no search was conducted, since the previously generated information would be used in that case. Additionally, other search refinements were also introduced. The most remarkable were: iterative deepening (which searches down to a certain level of the game tree), opening books (which include rules or move sequences that are known to be good to start a game), and endgame databases (which contain move sequences that are known to be good for ending a game, or even solutions to positions with a certain (small) number of pieces).

In 1975, Knuth [15] analyzed in detail the alpha-beta pruning algorithm. This algorithm uses a pruning technique which has the advantage of refraining from evaluating some nodes when unnecessary.

During the early $1980s$, chess programs based on microprocessors became reachable to a larger audience. However, this technology also made such programs very limited due to the small memory capabilities and the slow processors available at that time.

In 1989, two chess computers developed at Carnegie-Mellon University (*Hitech* and *Deep Thought*) were able to defeat a

human chessmaster each.

By 1996, the computer *Deep Thought*, which was able to examine 100 million chess positions per second, played a six-games match versus world chess champion Garry Kasparov. The final result of the match was 4 to 2 in favor of Kasparov.

In 1997, an updated version of *Deep Thought*, called *Deep Blue* was released. This computer was used again to play a six-game match against Garry Kasparov. This time, however, Kasparov lost the match (he obtained 2.5 points and *Deep Blue* obtained 3.5 points). After almost 50 years of research, the goal of having a computer that was able to defeat the chess world champion had finally been fulfilled.

In spite of its importance, this significant event did not stop the research in this area. The focus, however, changed towards the development of chess programs that did not require specialized hardware (as *Deep Blue* did). The advent of faster processors, cheaper memories and more elaborate search algorithms (including the use of metaheuristics) has made possible the development of very powerful chess engines that can run on conventional personal computers.

The main components of a chess engine are: (1) the move generator (it generates all possible moves at a given position on the board), (2) the search function (it finds the best variants in the search-tree from a given position on the board) and (3) the evaluation function. Due to the complexity of chess games it is only possible to represent the search tree down to a certain depth. Therefore, it is necessary to evaluate the terminal nodes of the search tree through the evaluation function, which is responsible for assigning a numerical value to a specific position on the board.

The evaluation function is the main component of any chess engine. Clearly, a successful implementation of the evaluation function allows a chess engine to play better, and, therefore, that is the focus of the work reported in this paper.

The evaluation function of chess engines relies on the use of weights (these weights are associated to the pieces involved in a certain position) to assess the numerical value of a certain board position. Developers of commercial chess engines normally carry out the tuning of these weights through lenghty trial-and-error processes (which make take years). Recently, however, the use of soft computing techniques (namely, evolutionary algorithms and neural networks) has allowed to tune these weights in a much faster (and still effective) manner.

The remainder of this paper is organized as follows. In Section II, we briefly present the previous related work. The chess engine adopted for our experiments is described in Section III. Our method is described in Section IV. In Section V, we present our experimental results. Finally, our conclusions and some possibe paths for future research are provided in Section VI.

II. Previous Related Work

A variety of evolutionary algorithms, including differential evolution [2], [3], [1], genetic algorithms [5], [6], genetic programming [11], [12] and evolutionary programming [7], [8], [17], [9], have been used for tuning the weights of the evaluation function of a chess engine.

Most of these previous works make use of co-evolution (tournaments between virtual players) to decide which virtual players will pass to the next generation, and only two of them have adopted grandmasters' games to decide which virtual player will pass to the following generation.

In [6], the authors used games from chess grandmasters in the objective function of their genetic algorithm. Additionally, the authors used co-evolution to improve the adjustment of the weights of their chess engine. In [20], Vázquez-Fernández et al. adjusted the weights of both the material values of the pieces and the mobility factor through an evolutionary algorithm. The work reported here, differs from this previous paper in that here, we adopt a larger number of weights (29, instead of the 5 adopted in [20]). Additionally, we incorporate here a direct search method (i.e., a mathematical programming technique that does not require gradient information): the Hooke-Jeeves method. This approach is used as a local search engine which improves the adjustment of the weights of the best virtual player obtained during the evolutionary process. As we will see later on, both our evolutionary algorithm and the local search engine are able to improve the rating of our chess engine, which is the main objective of our work.

III. Our Chess Engine

For carrying out our experiments, we developed a chess engine with the following characteristics:

- Alpha-beta algorithm [15].
- Stabilization of positions through the Quiescence algorithm that takes into account the exchange of material and king's checks.
- Use of iterative deepening and hash tables [4].
- Null-move heuristic.

Our chess program evaluates a given position on the board for a particular side, with the following expression:

$$eval = pV + mV \tag{1}$$

where:
pV is the sum of the positional values of chess pieces for a particular side.
mV is the sum of the material values of chess pieces for a particular side, and is given by the following expression.

$$mV = \sum_{i=1}^{r} X_i \tag{2}$$

where:
X_i represents the material value for piece i.
r is the number of pieces of one side in particular, regardless of the king ($r = 5$).

pV is given by the following expression:

$$pV = \sum_{i=1}^{s} P_i \tag{3}$$

where:
P_i represents the positional value for piece i.
s is the number of pieces of one side in particular ($s = 6$).

The king's positional value is given by:

$$P_{king} = \sum_{i=1}^{4} X_{king,i} * F_{king,i} \tag{4}$$

where:
$X_{king,i}$ is the weight of factor $F_{king,i}$. A *factor* is a positional characteristic of a particular piece; for example, its mobility, its column type, etc.
$F_{king,1}$ is the sum of material values of pieces that defend their king, i.e., those pieces whose movements act on its king's square or on its king's adjacent squares.
$F_{king,2}$ is the sum of material values of pieces that attack the king, i.e., those pieces whose movements act on its opposite king's square or on its opposite king's adjacent squares.
$F_{king,3}$ is true if the king is castled; otherwise, it is false.
$F_{king,4}$ is the number of pawns that protect their king.

The queen's positional value is given by (at the moment, let's consider only the queen's mobility):

$$P_{queen} = X_{queen,1} * F_{queen,1} \tag{5}$$

where:
$X_{queen,1}$ is the weight of factor F_i.
$F_{queen,1}$ is the queen's mobility.

The rook's positional value is given by:

$$P_{rook} = \sum_{i=1}^{7} X_{rook,i} * F_{rook,i} \tag{6}$$

where:
$X_{rook,i}$ is the weight of factor $F_{rook,i}$.
$F_{rook,1}$ is the rook's mobility.
$F_{rook,2}$ is true if on the rook's column there are no pawns; otherwise, it is false.
$F_{rook,3}$ is true if on the rook's column there are only adversary pawns; otherwise, it is false.
$F_{rook,4}$ is true if on the rook's column there are pawns for both sides and the rook is on front of its pawns; otherwise, it is false.
$F_{rook,5}$ is true if on the rook's column there are pawns for both sides and the rook is behind of its pawns; otherwise, it is false.
$F_{rook,6}$ is true if the rook is on the seventh row; otherwise, it is false.
$F_{rook,7}$ is true if there are at least two rooks on the seventh row; otherwise, it is false.

The bishop's positional value is given by:

$$P_{bishop} = X_{bishop,1} * F_{bishop,1} \tag{7}$$

where:
$X_{bishop,1}$ is the weight of factor $F_{bishop,1}$.
$F_{bishop,1}$ is the bishop's mobility.

The knight's positional value is given by:

$$P_{knight} = \sum_{i=1}^{7} X_{knight,i} * F_{knight,i} \tag{8}$$

where:
$X_{knight,i}$ is the weight of factor $F_{knight,i}$.
$F_{knight,1}$ is the mobility of the knight.
$F_{knight,2}$ is true if the knight is defended by a pawn; otherwise, it is false.
$F_{knight,3}$ is true if the knight cannot be evicted by an enemy pawn; otherwise, it is false.
$F_{knight,4}$ is true if the knight is in the squares $a_1, \ldots, a_8$, $b_1, \ldots, g_1$, $h_1, \ldots, h_8$, and $b_8, \ldots, g_8$ (which corresponds to the squares on the periphery of the board); otherwise, it is false.
$F_{knight,5}$ is true if the knight is in the squares $b_2, \ldots, b_7$, $c_2, \ldots, f_2$, $g_2, \ldots, g_7$, and $c_7, \ldots, f_7$; otherwise, it is false.
$F_{knight,6}$ is true if the knight is in the squares $c_3, \ldots, c_6$, d_3, e_3, $f_3, \ldots, f_6$, and $d_6, \ldots, e_6$; otherwise, it is false.
$F_{knight,7}$ is true if the knight is in the squares d_4, e_4, d_5, e_5; otherwise, it is false.

We expected that $X_{knight,4} < X_{knight,5} < X_{knight,6} < X_{knight,7}$ because if the knight is located in the center of the board its positional value will be better.

The pawn's positional value is given by:

$$P_{pawn} = \sum_{i=1}^{5} X_{pawn,i} * F_{pawn,i} \tag{9}$$

where:
$X_{pawn,i}$ is the weight of factor $F_{pawn,i}$.
$F_{pawn,1}$ is true if the pawn is doubled; otherwise, it is false.
$F_{pawn,2}$ is true if the pawn is isolated; otherwise, it is false.
$F_{pawn,3}$ is true if the pawn is backwards; otherwise, it is false.
$F_{pawn,4}$ is true if the pawn is central (i.e., if it is in $c4$, $c5$, $d4$, $d5$, $e4$, $e5$, $f4$ or $f5$ square); otherwise, it is false.
$F_{pawn,5}$ is true if the pawn is passed; otherwise, it is false.

The material value of the piece P_i corresponds to its static value. We assigned 300, 330, 500 and 900 points for the knight, bishop, rook and queen, respectively (as Shannon did in his work [18]). The material value for the pawn is 100.

The positional value of a piece is a dynamic value and depends on the characteristics of the position such as mobility, board location, strength, etc. In other works (Fogel et al. [7] for example) the piece's positional value is represented by *positional value tables*. The disadvantage of these methods is that the positional value of a piece is a static value (a knight in $f6$ square always has the same value regardless of their location and relationship with the other pieces). In this sense, one of the main ideas of our proposal is that the chess positional values depend directly on the characteristics of the position. Of course, while more features are taken into account in calculating the positional value of a piece, the positional value will be more accurate, and therefore, the position will be better evaluated.

The purpose of this work is to tune the weights of equations (2), (4), (5), (6), (7), (8) and (9) using evolutionary programming [10] and a database of chess grandmasters. The aim is that the adjustment of the weights performed by our approach leads to an increase in the rating of our chess engine.

IV. Our Proposed Approach

Our proposal consists of the following steps:

- **Exploration search**. It is the first step of our proposal, and is based on evolutionary programming [10] which has a selection mechanism based on a database of chess grandmaster games (supervised learning). The selection mechanism allows that the virtual players with more positions properly solved from a database of chess grandmaster games acquire the right to pass to the next generation. In our previous work [20], we conducted this phase in a similar way, but now we adjusted a larger number of weights (we went from 5 to 29 weights).
- **Exploitation search**. It is the second step of our proposal, and we carried out a local search procedure, aiming to improve the best virtual player obtained in the previous step. For that sake, we applied the Hooke-Jeeves algorithm to the best virtual player obtained from the evolutionary process. The objective function incorporated into the Hooke-Jeeves method also used a database of chess grandmaster games to carry out the adjustment of the weights under consideration.

Algorithm 1 EvolutionaryAlgorithm()

```
1: intializePopulation();
2: g = 0;
3: while g++< Gmax do
4:    scoreCalculation();
5:    selection();
6:    mutate();
7:    g++;
8: end while
```

The exploration search of our proposal was carried out with the evolutionary algorithm shown in Algorithm 1. The algorithm description is as follows. Line 1 initializes the weights of N virtual players with random values within their corresponding boundaries. Line 2 sets the generations counter equal to zero. Lines 3 to 8 carry out the adjusting of the weights for virtual players during $Gmax$ generations. In line 4, we calculate the score for each virtual player (in Algorithm 2 we will describe in more detail this aspect). In Line 5 we apply the selection mechanism so that only the best $N/2$ virtual players pass to the following generation. In line 6, we mutate the first half of the population in order to obtain the second half of the virtual players. That is, all the weights from each surviving parent were mutated to create one offspring (the weights that were mutated are shown in Table I). As done in [20], we adopted here Michalewicz's non-uniform mutation operator. Since we adopted evolutionary programming, no crossover operator is employed in our case. Finally, line 7 increases the generation counter by 1.

Algorithm 2 scoreCalculation()

```
1: for i = 0 → N − 1 do
2:    score[i] = 0;
3: end for
4: for each position p in database S do
5:    m = grandmasterMovement(p);
6:    setPosition(p);
7:    for each virtual player i do
8:       n = nextMovement(i);
9:       if m == n then
10:         score[i]++;
11:      end if
12:   end for
13: end for
```

The procedure for computing the score of each virtual player is described in Algorithm 2. In lines 1 to 3, we establish the score counter to zero for each virtual player. Line 4 chooses p training positions from database S. Line 5 chooses chess grandmaster movements for position p. Line 6 sets the position p (this allows to each virtual player to calculate its next movement). Finally, each virtual player calculates its next move n, and if this movement matches movement m, this virtual player increases its score by 1.

In the exploitation search, we employed the Hooke-Jeeves method to further adjust the weights of the best virtual player obtained during the exploration search step. The Hooke-Jeeves method is a direct search algorithm originally proposed in 1961 [13]. This method carries out a deterministic local search with a local descent algorithm, which does not make use of the objective function derivatives.

Algorithm 3 shows the method of Hooke-Jeeves. In this algorithm:

- x^0 is the best virtual player obtained in the exploratory search step.
- x^k represents the current virtual player.
- x^{k-1} represents the previous virtual player.
- x_p^{k+1} represents the pattern virtual player.
- x^{k+1} represents the next or new virtual player.

Algorithm 3 hookeJeeves()

Step 1. Define:
The initial virtual player $x^{(0)}$.
The increments Δ_i for each weight for $i = 1, \ldots, WN$.
The step reduction factor $\alpha > 1$.
A termination parameter $\epsilon > 0$.
Step 2. Perform exploratory search.
Step 3. Was the exploratory search successful (i.e, was a better virtual player found)?
Yes: Go to step 5.
No: Continue.
Step 4. Check for termination.
Is $||\Delta|| < \epsilon$?
Yes: Stop: return the best current virtual player.
No: Reduce the increments:
$\Delta_i = \Delta_i/\alpha$ for $i = 1, \ldots, WN$
Go to step 2.
Step 5. Perform patterns move.
$x_p^{k+1} = x^k + x^k - x^{k-1}$
Step 6. Perform exploratory search using x_p^{k+1} as the base virtual player; let the result be x^{k+1}.
Step 7. Is $f(x^{k+1}) > f(x^k)$
Yes: Set $x^{k-1} = x^k$; $x^k = x^{k+1}$
Go to step 5.
No: Go to step 4.

- WN is the number of weights for each virtual player.
- f is the objective function.

The objective function returns the number of positions solved by each virtual player. In our case, we randomly chose $M = 20$ positions from chess grandmaster games that were not solved by any virtual player during the exploratory search.

A. Initialization

During the exploratory search step, the initial population consisted of $N = 20$ virtual players (10 parents and 10 offspring in subsequent generations). Their weights (described in equations (2), (4), (5), (6), (7), (8) and (9)) were randomly generated with a uniform distribution within their allowable bounds (these bounds for each weight are shown in Table I).

B. Database of Games

In our experiments, we used a database consisting of 1000 games from chess grandmasters having a rating above 2600 Elo (see Appendix A). The games were taken from the Linares tournaments, from matches for the world chess championship, and from the Wijk aan Zee tournaments, among others.

V. Experimental Results

We carried out three experiments. The first experiment was based on the exploration and the exploitation stages of the search. In the second experiment, we performed matches between the virtual player obtained after the exploration phase and the virtual player obtained after the exploitation phase. Finally, in the third experiment, we carried out matches between our virtual players and the chess program *Chessmaster*.

TABLE I
Ranges of the weights for each virual player.

$Weight$	W_{low}	W_{high}
X_1 (PAWN_VALUE)	100	100
X_2 (KNIGHT_VALUE)	200	400
X_3 (BISHOP_VALUE)	200	400
X_4 (ROOK_VALUE)	400	600
X_5 (QUEEN_VALUE)	800	1000
$X_{king,1}$	0	4000
$X_{king,2}$	-4000	0
$X_{king,3}$	0	100
$X_{king,4}$	0	100
$X_{rook,1}$	0	100
$X_{rook,2}$	-50	50
$X_{rook,3}$	-50	50
$X_{rook,4}$	-50	50
$X_{rook,5}$	-50	50
$X_{rook,6}$	0	100
$X_{rook,7}$	0	100
$X_{bishop,1}$	0	100
$X_{knight,1}$	0	100
$X_{knight,2}$	0	100
$X_{knight,3}$	0	100
$X_{knight,4}$	-50	50
$X_{knight,5}$	-50	50
$X_{knight,6}$	-50	50
$X_{knight,7}$	-50	50
$X_{pawn,1}$	-50	50
$X_{pawn,2}$	-50	50
$X_{pawn,3}$	-50	50
$X_{pawn,4}$	-50	100
$X_{pawn,5}$	-50	100

In these experiments, our chess engine used the database *Olympiad.abk* in the opening phase. This database is included with the graphical user interface *Arena*[1]. In the following subsections we describe the experiments performed.

A. First experiment

The first experiment was divided in the two steps described in Section IV. In the first step, we applied exploration search to adjust the weights shown in Table I. In this case, we performed 30 runs, and in each of them, we used $Gmax = 200$ generations, $N = 20$ virtual players, and $p = 1000$ of training positions for chess grandmaster games. The best virtual player from these runs at generation 0, and at generation 200, were called $VP^0_{exploration}$, and $VP^{200}_{exploration}$, respectively.

Figure 1 shows the evolutionary process for the exploration search. The plot shows the number of positions solved (a total of 1000) for the best virtual player and the average weight values of the 20 virtual players during 200 generations. At generation 0, the number of positions solved for the average weight values was 187 (which corresponds to 18.7% of the positions), and 208 for the best virtual player (which corresponds to 20.8% of the positions). At generation 200, the number of positions solved for the average weight values

[1]http://www.playwitharena.com/

and the best virtual player was 328 (which corresponds to 32.8% of the positions). Note that this value is competitive with the value reported in [6]. At generation 200, the number of positions solved for the average weight values and the best virtual player was the same because we used Michalewicz's non-uniform mutation operator [16].

In the second step of the first experiment, we applied the exploitation search to the best virtual player obtained with the exploration search. In this case, we applied the Hooke-Jeeves algorithm with the following parameters:

- The step reduction factor $\alpha = 2$.
- The termination parameter $\epsilon = 0.5$.
- The increments $\Delta_i = 30$ for $i = 1, \ldots, WN$ (WN is the number of weights).

The second column of the Table II shows the tuning of weights for the virtual player $VP^{200}_{exploration}$ (the best virtual player obtained after the exploration search). In the third column, we show the tuning of weights for the virtual player $VP_{exploitation}$ (the virtual player obtained after the exploitation search). In both cases, we can see that the material values of the pieces are close to their "theoretical" values.

Next, we used the resulting weights of the virtual player $VP_{exploitation}$ to test them with the 1000 training positions from chess grandmaster games. In this case, the virtual player $VP_{exploitation}$ successfully resolved 483 of the 1000 positions (which corresponds to 48.3%). Therefore, we can see that the number of positions solved using both exploration and exploitation was larger than when we used only exploration (from 32.8% to 48.3%). We consider that the number of positions solved by our method was satisfactory because we used only a depth of one ply in the search tree. It is noteworthy that David-Tabibi et al. [6] also used one ply in their work.

With the completion of the exploration and exploitation search, we used an additional 1000 positions for testing the virtual player $VP_{exploitation}$. We let this virtual player perform a 1-ply search on each of these positions, and the percentage of correctly solved positions was 47.9%. This indicates that the first 1000 positions used for training cover most of the types of positions that can arise.

B. Second experiment

In this experiment, our chess engine used a search depth of four ply. We carried out 200 games between the virtual player $VP^{200}_{exploration}$ and the virtual player $VP_{exploitation}$ (each virtual player played 100 games with black pieces and 100 with white pieces). The virtual player $VP_{exploitation}$ won, drew, and lost 142, 25, and 32 games, respectively, versus the virtual player $VP^{200}_{exploration}$.

Next, we used the Bayeselo tool[2] to estimate the ratings of our chess engine using a minorization-maximization algorithm [14]. The obtained ratings are shown in Table III. In this table, we can see that the rating for the virtual player $VP_{exploitation}$ was 2425, and the rating for the virtual player $VP^{200}_{exploration}$ was 2205, representing an increase of 220 rating points between the virtual player obtained with exploration plus exploitation search and the virtual player obtained only with exploration search. In this table the absolute level for the Bayeselo tool was set in 2200 rating points.

[2]http://remi.coulom.free.fr/Bayesian-Elo/

TABLE II
VALUES OF THE WEIGHTS AFTER THE EXPLORATION SEARCH (SHOWN IN THE SECOND COLUMN) AND AFTER THE EXPLOITATION SEARCH (SHOWN IN THE THIRD COLUMN).

$Weight$	$VP^{200}_{exploration}$	$VP_{exploitation}$
X_1 (PAWN_VALUE)	100	100
X_2 (KNIGHT_VALUE)	297	302
X_3 (BISHOP_VALUE)	315	319
X_4 (ROOK_VALUE)	502	506
X_5 (QUEEN_VALUE)	923	910
$X_{king,1}$	1650	1675
$X_{king,2}$	-1430	-1425
$X_{king,3}$	47	45
$X_{king,4}$	65	72
$X_{rook,1}$	62	73
$X_{rook,2}$	45	46
$X_{rook,3}$	27	33
$X_{rook,4}$	32	27
$X_{rook,5}$	-8	-7
$X_{rook,6}$	63	68
$X_{rook,7}$	78	82
$X_{bishop,1}$	72	76
$X_{knight,1}$	64	68
$X_{knight,2}$	56	72
$X_{knight,3}$	52	73
$X_{knight,4}$	-12	-15
$X_{knight,5}$	03	6
$X_{knight,6}$	15	26
$X_{knight,7}$	42	43
$X_{pawn,1}$	-32	-44
$X_{pawn,2}$	-47	-48
$X_{pawn,3}$	-44	-41
$X_{pawn,4}$	43	48
$X_{pawn,5}$	48	49

TABLE III
RATINGS OF THE SECOND EXPERIMENT.

Name	**Elo**	**+**	**-**	**Games**	**Score (%)**	**Oppo.**
$VP_{exploitation}$	2425	25	24	200	77%	2205
$VP^{200}_{exploration}$	2205	24	25	200	23%	2425

We can have an idea of the playing strength of our virtual players using the classification of the United States Chess Federation (see Appendix A). From Table V, we can see that the strength of the virtual player $VP^{200}_{exploration}$ (2205 rating points) is at the level of a master in chess, and the strength of the virtual player $VP_{exploitation}$ (2425 rating points) is at the level of a senior master in chess.

C. Third experiment

In this experiment, we carried out 200 games among the virtual players $VP^{0}_{exploration}$, $VP^{200}_{exploration}$, $VP_{exploitation}$, and the popular chess program *Chessmaster* (grandmaster edition) which was set at 2500 rating points. The results are shown in Figure 2. In this figure, we can see that $Chessmaster_{2500}$'s

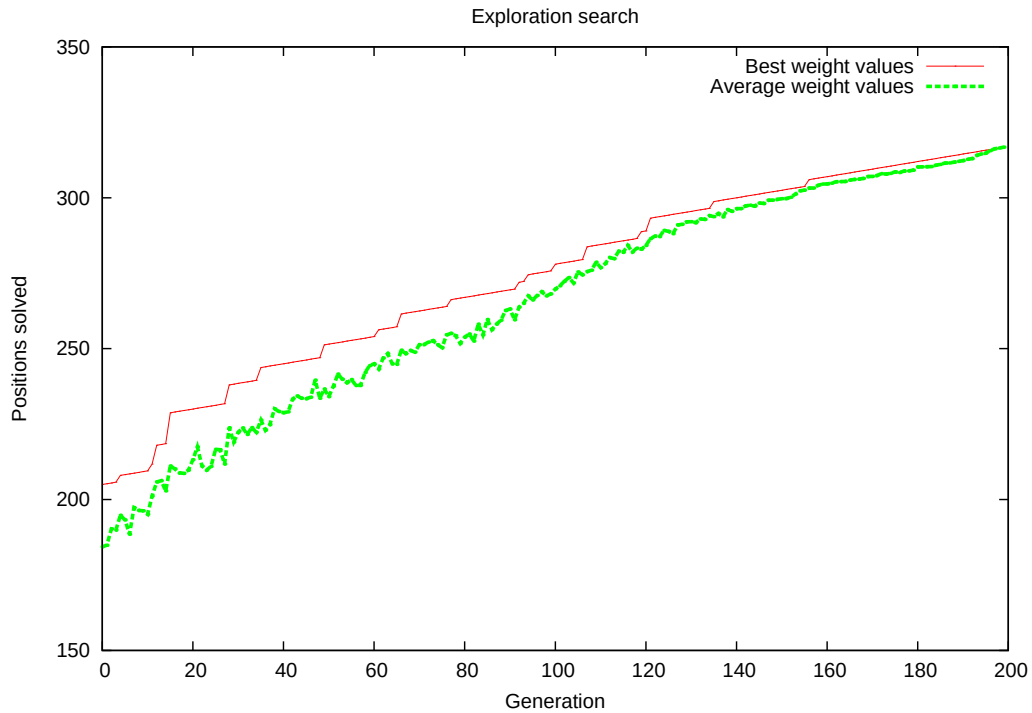

Fig. 1. Evolutionary process for the exploration search. The plot shows the number of positions solved (a total of 1000) for the best virtual player and the average weight values of the 20 virtual players during 200 generations.

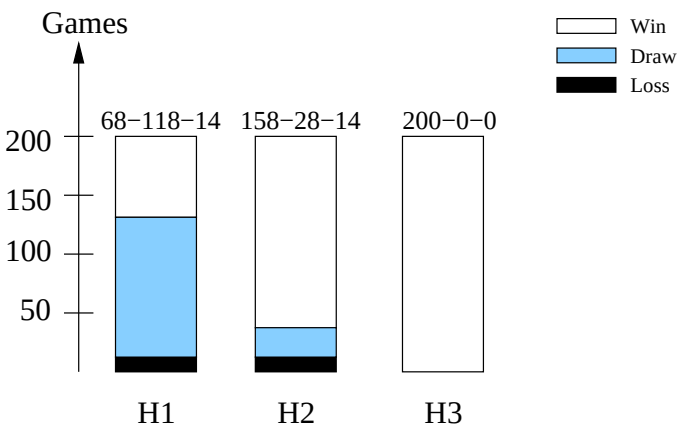

Fig. 2. Histogram of wins, draws and losses for $Chessmaster_{2500}$ against $VP_{exploitation}$ (H1), $VP^{200}_{exploration}$ (H2), $VP^{0}_{exploration}$ (H3).

wins, draws, and losses were 68, 118, and 14, respectively, versus the virtual player $VP_{exploitation}$ (denoted as the histogram H1 in the Figure). Also, $Chessmaster_{2500}$'s wins, draws, and losses, were 158, 28, and 14, respectively, versus the virtual player $VP^{200}_{exploration}$ (denoted as the histogram H2 in the Figure), respectively, and so on.

Based on these played games, we used again the Bayeselo tool to estimate the ratings of $Chessmaster_{2500}$ (CM_{2500} in Table IV), and the virtual players $VP_{exploitation}$, $VP^{200}_{exploration}$, and $VP^{0}_{exploration}$. The obtained ratings are shown in Table IV. In this table we can see that the rating for the virtual players $VP^{0}_{exploration}$, $VP^{200}_{exploration}$, and $VP_{exploitation}$ were 1600, 2197, and 2424, respectively. In this experiment, our chess engine used a search depth of six ply. In this table the absolute level for the Bayeselo tool was set in 2500 rating points.

TABLE IV
RATINGS OF THE THIRD EXPERIMENT.

Name	Elo	+	-	Games	Score (%)	Oppo.	Draws (%)
CM_{2500}	2499	23	23	600	83%	2074	24%
$VP_{exploitation}$	2424	28	28	200	37%	2499	59%
$VP^{200}_{exploration}$	2197	39	43	200	14%	2499	14%
$VP^{0}_{exploration}$	1600	144	348	200	0%	2499	0%

VI. CONCLUSIONS AND FUTURE WORK

In this work, we used two steps to carry out the tuning of the weights of a chess engine. In the first step, we performed an exploration search through an evolutionary algorithm with supervised learning. The selection mechanism of our evolutionary algorithm used games from chess grandmasters to decide which virtual player would pass to the following generation. This step is similar to our previous work presented at CEC'2011 with the difference that now we adjusted a larger number of weights (form 5 to 29 weights). With this increase in the number of weights, we obtained an increase in the rating of our chess engine, since we went from 1463 to 2205 (the value of 1463 was obtained in 10 games against a human player who has 1737 rating points). We believe that

TABLE V
ELO RATING SYSTEM

Interval	Level
2400 and above	Senior Master
$2200 - 2399$	Master
$2000 - 2199$	Expert
$1800 - 1999$	Class A
$1600 - 1799$	Class B
$1400 - 1599$	Class C
$1200 - 1399$	Class D
$1000 - 1199$	Class E

this confirms the proper working of our evolutionary algorithm in adjusting weights of our chess engine.

In the second step, we used the Hooke-Jeeves algorithm to continue the adjustment of the weights for the best virtual player obtained in the previous step. Using this algorithm as a local search engine, we increased the rating of our chess engine from 2205 to 2425 points (in the second experiment), and from 2197 to 2424 points (in the third experiment). Therefore, we conclude that the local search procedure based on the Hooke-Jeeves algorithm was successful.

On the other hand, we can see that the number of positions solved using exploration plus exploitation search was larger than when we only used exploration search (from 32.8% to 48.3%). We believe that the number of positions solved by our method was satisfactory because we used only a depth of one ply in the search tree. Furthermore, this value is greater than that obtained in the previous related work.

As part of our future work, we plan to adjust more weights with the idea of creating a chess program that is able to play better. We also plan to perform more experiments varying the population size and increasing the number of games in the chessmasters database. In order to extend the search tree depth we will plan to add extensions to the quiescence algorithm, for example, pawn promotions or pawns on seventh row. Finally, and considering that support vector machines are a standard supervised learning method, as part of our future work, we plan to use them to carry out the exploration search.

ACKNOWLEDGEMENTS

The first author acknowledges support from CINVESTAV-IPN, CONACyT and the National Polytechnical Institute (IPN) to pursue graduate studies at the Computer Science Department of CINVESTAV-IPN. The second author acknowledges support from CONACyT project no. 103570.

APPENDIX A

The Elo rating system is a method for calculating the relative strength of players in games with two opponents like chess, association football, American college football, and basketball, among others. This method was created by the mathematician Arpad Elo, and has been adopted by the USCF (United States Chess Federation) since 1960 and by the FIDE (Fédération Internationale des Échecs) since 1970. In Table V we show the classification of the USCF.

REFERENCES

[1] B. Bošković, J. Brest, A. Zamuda, S. Greiner, and V. Žumer. History Mechanism Supported Differential Evolution for Chess Evaluation Function Tuning. *Soft Computing - A Fusion of Foundations, Methodologies and Applications*, 2011. (in press).

[2] B. Bošković, S. Greiner, J. Brest, and V. Žumer. A differential evolution for the tuning of a chess evaluation function. In *2006 IEEE Congress on Evolutionary Computation*, pages 1851–1856, Vancouver, BC, Canada, July 16-21 2006. IEEE Press.

[3] B. Bošković, S. Greiner, J. Brest, A. Zamuda, and V. Žumer. An Adaptive Differential Evolution Algorithm with Opposition-Based Mechanisms, Applied to the Tuning of a Chess Program. In U. Chakraborty, editor, *Advances in Differential Evolution*, pages 287–298. Springer, Studies in Computational Intelligence, Vol. 143, Heidelberg, Germany, 2008.

[4] D. Breuker, J. W. H. M. Uiterwijk, and H. J. V. D. Herik. Information in transposition tables. *Advances in Computer Chess 8*, pages 199–211, 1997.

[5] O. David-Tabibi, M. Koppel, and N. S. Netanyahu. Expert-driven genetic algorithms for simulating evaluation functions. *Genetic Programming and Evolvable Machines*, 12:5–22, March 2011.

[6] O. David-Tabibi, H. J. van den Herik, M. Koppel, and N. S. Netanyahu. Simulating human grandmasters: evolution and coevolution of evaluation functions. In *GECCO'09*, pages 1483–1490, 2009.

[7] D. B. Fogel, T. J. Hays, S. L. Hahn, and J. Quon. A self-learning evolutionary chess program. *Proceedings of the IEEE*, 92(12):1947–1954, 2004.

[8] D. B. Fogel, T. J. Hays, S. L. Hahn, and J. Quon. Further evolution of a self-learning chess program. In *Proceedings of the 2005 IEEE Symposium on Computational Intelligence and Games (CIG05)*, pages 73–77, Essex, UK, April 4-6 2005. IEEE Press.

[9] D. B. Fogel, T. J. Hays, S. L. Hahn, and J. Quon. The blondie25 chess program competes against fritz 8.0 and a human chess master. In S. J. Louis and G. Kendall, editors, *Proceedings of the 2006 IEEE Symposium on Computational Intelligence and Games (CIG06)*, pages 230–235, Reno, Nevada, USA, May 22-24 2006. IEEE Press.

[10] L. J. Fogel. *Artificial Intelligence through Simulated Evolution*. John Wiley, New York, 1966.

[11] A. Hauptman. Gp-endchess: Using genetic programming to evolve chess endgame players. In *In: Proceedings of 8th European Conference on Genetic Programming (EuroGP2005*, pages 120–131. Springer, 2005.

[12] A. Hauptman and M. Sipper. Evolution of an efficient search algorithm for the mate-in-n problem in chess. In *Proceedings of the 10th European conference on Genetic programming*, EuroGP'07, pages 78–89, Berlin, Heidelberg, 2007. Springer-Verlag.

[13] R. Hooke and T. A. Jeeves. " direct search" solution of numerical and statistical problems. *J. ACM*, 8:212–229, April 1961.

[14] R. Hunter. Mm algorithms for generalized bradley-terry models. *The Annals of Statistics*, 32:2004, 2004.

[15] D. E. Knuth and R. W. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6(4):293–326, 1975.

[16] Z. Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs*. Springer-Verlag, second edition, 1996.

[17] H. Nasreddine, H. Poh, and G. Kendall. Using an Evolutionary Algorithm for the Tuning of a Chess Evaluation Function Based on a Dynamic Boundary Strategy. In *Proceedings of 2006 IEEE international Conference on Cybernetics and Intelligent Systems (CIS'2006)*, pages 1–6. IEEE Press, 2006.

[18] C. Shannon. Programming a computer for playing chess. *Philosophical Magazine*, 7(41):256–275, 1950.

[19] A. Turing. *Digital Computers Applied to Games, of Faster than Thought*, chapter 25, pages 286–310. Pitman, 1953.

[20] E. Vázquez-Fernández, C. A. C. Coello, and F. D. S. Troncoso. An evolutionary algorithm for tuning a chess evaluation function. In *2011 IEEE Congress on Evolutionary Computation*, New Orleans, Louisiana, USA, June 5–8 2011.