Applied Soft Computing 13 (2013) 3234–3247

Contents lists available at SciVerse ScienceDirect

Applied Soft Computing

journal homepage: www.elsevier.com/locate/asoc

An evolutionary algorithm with a history mechanism for tuning a chess evaluation function

Eduardo Vázquez-Fernández [a,*], Carlos A. Coello Coello [a], Feliú D. Sagols Troncoso [b]

[a] CINVESTAV-IPN (Evolutionary Computation Group), Computer Science Department, Av. IPN No. 2508, México, D.F. 07300, Mexico
[b] CINVESTAV-IPN, Departamento de Matemáticas, Av. IPN No. 2508, Col. San Pedro Zacatenco, México, D.F. 07360, Mexico

ARTICLE INFO

Article history:
Received 24 October 2011
Received in revised form 9 May 2012
Accepted 24 February 2013
Available online 16 March 2013

Keywords:
Evolutionary algorithm
Evolutionary programming
Chess engine
Artificial intelligence
Chess evaluation function

ABSTRACT

Here, we propose an evolutionary algorithm (i.e., evolutionary programming) for tuning the weights of a chess engine. Most of the previous work in this area has normally adopted co-evolution (i.e., tournaments among virtual players) to decide which players will pass to the following generation, depending on the outcome of each game. In contrast, our proposed method uses evolution to decide which virtual players will pass to the next generation based on the number of positions solved from a number of chess grandmaster games. Using a search depth of 1-ply, our method can solve 40.78% of the positions evaluated from chess grandmaster games (this value is higher than the one reported in the previous related work). Additionally, our method is capable of solving 53.08% of the positions using a historical mechanism that keeps a record of the "good" virtual players found during the evolutionary process. Our proposal has also been able to increase the competition level of our search engine, when playing against the program *Chessmaster* (grandmaster edition). Our chess engine reached a rating of 2404 points for the best virtual player with supervised learning, and a rating of 2442 points for the best virtual player with unsupervised learning. Finally, it is also worth mentioning that our results indicate that the piece material values obtained by our approach are similar to the values known from chess theory.

1. Introduction

The origins of computer chess date back to the pioneering efforts of Alan Turing and Claude Shannon in the mid and late 1940s. In 1947, Alan Turing [25] designed a program to play chess and, in 1949, Claude Shannon [22] proposed two strategies to implement a chess engine. The first of them, called "Type A," considered all possible moves to a fixed depth of the search tree, and the second, called "Type B," used chess knowledge to explore the main lines to a greater depth.

During the 1950s, chess programs played at a very basic level, but by the 1960s, chess programs could defeat amateur chess players. During the 1970s, chess programs began to use heuristics and specialized hardware to improve their rating. During the 1980s, tournaments between chess programs and humans started.

During the 1990s, chess programs became powerful enough to start challenging chess masters. In 1997, *Deep Blue* defeated the world chess champion, Garry Kasparov, with a final score of 3.5 to 2.5 (Deep Blue was capable of evaluating 200 million positions per second). In 2002, Garry Kasparov had a six games match against the chess program that was the world champion at that time, *Deep Junior*. The final score was 3 to 3. In 2006, the world chess champion Vladimir Kramnik, from Russia, was defeated by the program *Deep Fritz*. The final score was 4 to 2 in favor of *Deep Fritz*.

Adjusting weights of a chess engine is an interesting problem because developers of commercial chess programs must fine-tune the weights of their evaluation functions using exhaustive test procedures, so that they can be improved as much as possible. However, a manual fine-tuning of weights is a difficult and time consuming process, and therefore the need to automate this task.

The current paper is an extended and improved contribution of our previous work [27]. Here, we propose an evolutionary algorithm (namely, evolutionary programming) to tune the weights of a chess evaluation function through a database of chess grandmaster games. The main difference with our previous work is that here, we tune a larger number of weights. Furthermore, we also add a historic mechanism that allows to retain "good" virtual players during the evolutionary process. In this work, we used 33 weights, which is a value similar to the one adopted by other authors (e.g., David-Tabibi et al. [8] used 35 weights).

The remainder of this paper is organized as follows. In Section 2, we briefly review the previous related work that makes use of co-evolution (tournaments among virtual players), and the only two works that we could find in the specialized literature, which

* Corresponding author. Tel.: +52 55 5747 3800x6564; fax: +52 55 5747 3757.
E-mail addresses: eduardovf@hotmail.com, edvazquezf@ipn.mx (E. Vázquez-Fernández), ccoello@cs.cinvestav.mx (C.A.C. Coello), fsagols@math.cinvestav.edu.mx (F.D.S. Troncoso).

1568-4946/$ – see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.asoc.2013.02.015

make use of evolution through a database of grandmaster games. In Section 3, we describe the characteristics of our chess engine, as well as the mathematical expression adopted for the evaluation function of our chess engine. In Section 4, we show the methodology that we adopted to tune the weights of our chess engine. In Section 5, we present our experimental results. Finally, our conclusions and some possible paths for future research are provided in Section 6.

2. Previous related work

First, we will briefly discuss works that do not make use of an evolutionary algorithm for tuning the weights of the evaluation function of a chess engine. Thrun [24] developed the program NeuroChess which learned to play chess from final outcomes with an evaluation function represented by neural networks. This work also included both temporal difference learning [23] and explanation-based learning [9]. Hsu et al. [16] tuned the weights of their evaluation function for the computer *Deep Thought* (later called *Deep Blue*) using a database of grandmaster-level games. Beal and Smith [1] determined the values of the pieces of a chess game using temporal-difference learning. In further work, the concept of piece-square values was introduced into their work [2].

Evolutionary algorithms have also been used before for tuning the evaluation function of a chess engine. Kendall and Whitwell [18] used them for tuning the evaluation function of their chess program. Nasreddine et al. [21] proposed a real-coded evolutionary algorithm that incorporated the so-called "dynamic boundary strategy" where the boundaries of the interval of each weight are dynamic.

Bošković presented three works that make use of differential evolution to adjust the weights of the evaluation function of their chess program. In their first work [4], they tuned the chess material values and the mobility factor of the evaluation function. The weights obtained matched the values known from chess theory. In a second work, Bošković et al. [5] employed adaptation and opposition-based optimization mechanisms with co-evolution to improve the rating of their chess program. In a third work, Bošković et al. [3] improved their opposition-based optimization mechanisms with a new history mechanism which uses an auxiliary population containing competent individuals. This mechanism ensures that skilled individuals are retained during the evolutionary process.

Genetic programming has also been used for tuning the weights of the chess evaluation function. Hauptman and Sipper [14] evolved strategies for playing chess end-games. Their evolved program could draw against CRAFTY which is a state-of-the-art chess engine having a rating of 2614 points. In a second work, Hauptman and Sipper [15] evolved entire game-tree search algorithms to solve mate-in-N problems in which the opponent cannot avoid being mated in at most N moves. It is worth noticing that this work does not adopt the alpha-beta pruning algorithm.

Genetic algorithms have also been used for tuning the weights of the chess evaluation function. David-Tabibi et al. [7] used reverse engineering to adjust the weights of an evaluation function. Basically, they used a grandmaster-level chess program for tuning the weights of their chess program. In a second work, David-Tabibi et al. [8] combined supervised and unsupervised learning to build a grandmaster-level program. This work presented the first attempt to adjust the weights of a chess program by learning only from a database of games played by humans.

Evolutionary programming has been used before by other authors for tuning the weights of the chess evaluation function. Fogel et al. [10] used this sort of evolutionary algorithm to improve the rating of a chess program by 400 points. They tuned the material values of the pieces, the piece-square values, and the weights of three neural networks. Their computer program learned chess by playing games against itself. In a second work, Fogel et al. [11] incorporated co-evolution, but this time, they evolved their program during 7462 generations, reaching a rating of 2650. The resultant program was called *Blondie25*. In a third work, Fogel et al. [12], used rules for managing the time allocated per move inside their program *Blondie25*. They achieved a rating of 2635 points against the program *Fritz*8.0, which was rated #5 in the world. It is noteworthy that *Blondie25* was also the first machine learning based chess program able to defeat a human chess master.

Vázquez-Fernández et al. [26] used a database of typical chess problems to adjust the weights of the evaluation function of their chess engine. Using evolutionary programming as their search engine, they mutated only those weights involved in the solution of the current problem and adapted the mutation mechanism through the number of problems solved by each virtual player. With their algorithm they obtained the "theoretical" values of the pieces and achieved an increase in the strength of their chess engine of 335 points. In a further paper, Vázquez-Fernández et al. [27] used a database of games played by chess grandmasters to adjust the weights of the material values and the mobility factor of the pieces. In this case, they obtained the "theoretical" values of the pieces with their evolutionary algorithm.

To the authors' best knowledge, there are only two works in which the selection mechanism of an evolutionary algorithm used to play chess is based on databases of chess grandmaster to decide which virtual players will pass to the following generation (in the remaining works in which evolutionary algorithms were adopted in some way, the proposed approaches used the final results of a game: win, loss or draw). In the first of them [8], the authors carried out (supervised) learning using a genetic algorithm. Additionally, in that paper, the authors used co-evolution (unsupervised learning) to improve the adjustment of the weights of their chess engine. In the second work [27], the authors adjusted the weights of both the material values of the pieces and the mobility factor through an evolutionary algorithm. Our work differs from this last paper in that we adopt here a larger number of weights. Additionally, we also adopt a historic mechanism to allow the best virtual players to survive throughout the evolutionary process.

3. Our chess engine

To carry out our experiments, we developed a chess engine with the following characteristics:

- The search depth adopted by our engine is of one ply (which corresponds to the movement of one side) for the training phase as used in [8], and of six ply for the games among virtual players as recommended in [16].
- We used the alpha-beta pruning search algorithm [19].
- We incorporated a mechanism to stabilize positions through the Quiescence algorithm, which takes into account the exchange of material and the king's checks.
- We used hash tables and iterative deepening [6]. A Hash Table is a data structure that allows to save the value of a given position on the board with the purpose of not having to calculate it again. Furthermore, chess engines do not perform your search with the alpha-beta algorithm to a fixed depth, instead they use a technique called iterative deepening. With this technique a chess engine searches at a depth of two, three, four, and so on, with the purpose that after finishing the search time, a catastrophic movement does not happen.

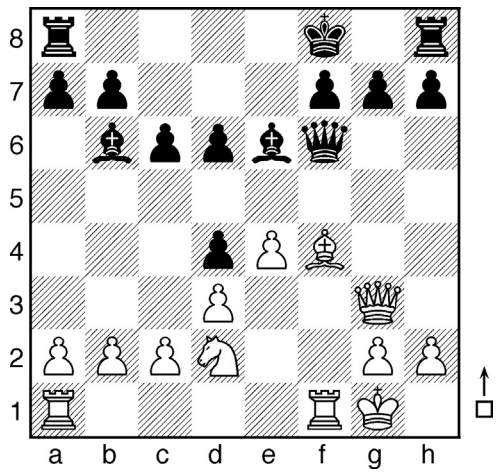

Fig. 1. Example diagram to illustrate feature extraction.

Our chess program evaluates the position of the virtual player A with the following expression:

$$eval = \sum_{i=1}^{22} Weight_i * f_i \quad (1)$$

where $Weight_i$ is one of the weights shown in Table 1, and f_i is the feature of the weight $Weight_i$ for the virtual player A. The description of the features are the following:

- f_{PAWN_VALUE} is the number of pawns of the virtual player A. For example, in Fig. 1 the white player has five pawns.
- f_{KNIGHT_VALUE} is the number of knights of the virtual player A. For example, in Fig. 1 the white player has one knight.
- f_{BISHOP_VALUE} is the number of bishops of the virtual player A. For example, in Fig. 1 the white player has one bishop.
- f_{ROOK_VALUE} is the number of rooks of the virtual player A. For example, in Fig. 1 the white player has two rooks.
- f_{QUEEN_VALUE} is the number of queens of the virtual player A. For example, in Fig. 1 the white player has one queen.
- $f_{PAWN\_DOUBLED\_PENALTY\_VALUE}$ denotes the number of doubled pawns of the virtual player A. For example, in Fig. 1 the white player has zero doubled pawns.
- $f_{PAWN\_ISOLATED\_PENALTY\_VALUE}$ denotes the number of isolated pawns of the virtual player A. For example, in Fig. 1 the white player has zero isolated pawns.
- $f_{PAWN\_BACKWARD\_PENALTY\_VALUE}$ denotes the number of backward pawns of the virtual player A. For example, in Fig. 1 the white player has zero backward pawns.
- f_{PAWN_PASSED} denotes the number of passed pawns of the virtual player A. For example, in Fig. 1 the white player has zero passed pawns.
- $f_{PAWN\_CENTRAL}$ denotes the number of central pawns of the virtual player A (pawns located on squares $c4$, $c5$, $d4$, $d5$, $e4$, $e5$, $f4$ or $f6$). For example, in Fig. 1 the white player has one central pawn.
- $f_{KNIGHT\_SUPPORTED}$ denotes the number of knights (of the virtual player A) supported or defended by one of his pawns. For example, in Fig. 1 the white player has zero knights supported.
- $f_{KNIGHT\_OPERATIONS\_BASE}$ denotes the number of knights (of the virtual player A) in an operation's base (it is when a knight cannot be evicted from its position by an opponent's pawn). For example, in Fig. 1 the white player has zero knights in a operation's base.
- $f_{KNIGHT\_PERIPHERY\_0}$ denotes the number of knights (of the virtual player A) in the squares $a_1, \ldots, a_8$, $b_1, \ldots, g_1$, $h_1, \ldots, h_8$, and $b_8, \ldots, g_8$. For example, in Fig. 1 the white player has zero knights in *periphery* _ 0.
- $f_{KNIGHT\_PERIPHERY\_1}$ denotes the number of knights (of the virtual player A) in the squares $b_2, \ldots, b_7$, $c_2, \ldots, f_2$, $g_2, \ldots, g_7$, and $c_7, \ldots, f_7$. For example, in Fig. 1 the white player has one knight in *periphery* _ 1.
- $f_{KNIGHT\_PERIPHERY\_2}$ denotes the number of knights (of the virtual player A) in the squares $c_3, \ldots, c_6$, $d_3, e_3, f_3, \ldots, f_6$, and $d_6, \ldots, e_6$. For example, in Fig. 1 the white player has zero knights in *periphery* _ 2.
- $f_{KNIGHT\_PERIPHERY\_3}$ denotes the number of knights (of the virtual player A) in the squares d_4, e_4, d_5, e_5. For example, in Fig. 1 the white player has zero knights in *periphery* _ 3.
- $f_{ROOK\_OPEN\_COLUMN}$ denotes the number of rooks (of the virtual player A) in an open column (a column without pawns). For example, in Fig. 1 the white player has zero rooks in a open column.
- $f_{ROOK\_SEMIOPEN\_COLUMN}$ denotes the number of rooks (of the virtual player A) in a semi-open column (a column that contains only opponent's pawns). For example, in Fig. 1 the white player has one rook in a open column.
- $f_{ROOK\_CLOSED\_COLUMN\_BEHIND}$ denotes the number of rooks (of the virtual player A) in a closed column behind of its pawns. We defined a closed column as the column that contains pawns of both players. For example, in Fig. 1 the white player has one rook in a closed column behind of its pawns.
- $f_{ROOK\_CLOSED\_COLUMN\_AHEAD}$ denotes the number of rooks (of the virtual player A) in a closed column ahead of its pawns. For example, in Fig. 1 the white player has zero rooks in a closed column ahead of its pawns.
- $f_{ROOK\_SEVEN\_RANK}$ denotes the number of rooks (of the virtual player A) in the seventh rank. For example, in Fig. 1 the white player has zero rooks in the seventh rank.
- $f_{KNIGHT\_MOBILITY}$ is the number of knights' moves of the virtual player A. For example, in Fig. 1 the knight's moves for the white player is 4.
- $f_{BISHOP\_MOBILITY}$ is the number of bishops' moves of the virtual player A. For example, in Fig. 1 the bishops' moves for the white player is 5.
- $f_{ROOK\_MOBILITY}$ is the number of rooks' moves of the virtual player A. For example, in Fig. 1 this value is 4 for white rook on $a1$.
- $f_{QUEEN\_MOBILITY}$ is the number of queens' moves of the virtual player A. For example, in Fig. 1 the queens' moves for the white player is 10.
- $f_{KING\_MOBILITY}$ is the number of king's moves of the virtual player A. For example, in Fig. 1 the king's moves for the white player is 2.
- $f_{KING\_ATTACKING\_MATERIAl}$ is the sum of the material value of the pieces that are attacking the opposite king. We mean those pieces whose movements act on its opposite king's square or on its opposite king's adjacent squares.
- $f_{KING\_DEFENDING\_MATERIAl}$ is the sum of the material value of the pieces that are defending its king. We mean those pieces whose movements act on its opposite king's square or on its opposite king's adjacent squares.
- $f_{KING\_CASTLING}$ is a binary value. It is true if the king is castled; otherwise, it is false. For example, in Fig. 1 this value is true for the white king.
- f_{KING_PAWNS} is the number of pawns located on its king's adjacent squares. For example, in Fig. 1 this value is two for the white king.
- f_{BISHOP_AHEAD} is the number of pawns which are in front of its bishop and obstructing its movement. For example, in Fig. 1 this value is one for the black bishop on $b6$.
- $f_{BISHOP\_PAWNS\_MOBILITY}$ is the number of movements of the pawns which obstruct the movement of the bishop. For example, in Fig. 1 this value is zero for the black bishop on $b6$.

Table 1
Ranges of the weights adopted in our approach and final weights for the virtual player $VP^{33}_{200,10}$.

Number	Weight	W_{low}	W_{high}	Weights for $VP^{33}_{200,10}$
1	PAWN_VALUE	100	100	100.0
2	KNIGHT_VALUE	200	400	295.2
3	BISHOP_VALUE	200	400	315.4
4	ROOK_VALUE	400	600	489.7
5	QUEEN_VALUE	800	1000	921.3
6	PAWN_DOUBLED_PENALTY_VALUE	−50	50	−14.3
7	PAWN_ISOLATED_PENALTY_VALUE	−50	50	−23.2
8	PAWN_BACKWARD_PENALTY_VALUE	−50	50	−19.5
9	PAWN_PASSED	−50	100	34.3
10	PAWN_CENTRAL	−50	100	19.7
11	KNIGHT_SUPPORTED	−50	100	16.2
12	KNIGHT_OPERATIONS_BASE	−50	100	15.3
13	KNIGHT_PERIPHERY_0	−50	50	−10.3
14	KNIGHT_PERIPHERY_1	−50	50	14.1
15	KNIGHT_PERIPHERY_2	−50	50	19.7
16	KNIGHT_PERIPHERY_3	−50	50	26.4
17	ROOK_OPEN_COLUMN	−50	50	22.1
18	ROOK_SEMIOPEN_COLUMN_BEHIND	−50	50	−5.7
19	ROOK_CLOSED_COLUMN	−50	50	−11.3
20	ROOK_SEVEN_RANK	−50	50	49.6
21	KNIGHT_MOBILITY	0	100	27.8
22	BISHOP_MOBILITY	0	100	34.1
23	ROOK_MOBILITY	0	100	37.2
24	QUEEN_MOBILITY	0	100	12.3
25	KING_MOBILITY	0	100	5.1
26	ROOK_SEMIOPEN_COLUMN_AHEAD	−50	50	9.3
27	KING_ATTACKING_MATERIAl	−100	0	−67.8
28	KING_DEFENDING_MATERIAl	0	100	52.3
29	KING_CASTLING	0	100	51.2
30	KING_PAWNS	0	100	49.3
31	BISHOP_AHEAD	−100	0	39.6
32	BISHOP_PAWNS_MOBILITY	0	100	21.3
33	BISHOP_PAIR	0	100	27.0

- f_{BISHOP_PAIR} is a binary value. It is true if the player A has the bishop pair; otherwise, it is false. For example, in Fig. 1 this value is true for the black side.

The main aim of the work reported here is to show that the weights of the evaluation function from Eq. (1) can be tuned using evolutionary programming [13]. In our approach, the training of the virtual players is conducted using a database of chess grandmaster games. Additionally, our work proposes the use of a historic mechanism which allows good virtual players to survive throughout the evolutionary process.

4. Evolutionary algorithm with a history mechanism for tuning a chess evaluation function

As indicated before, our proposed approach is based on an evolutionary algorithm (evolutionary programming [13]) which has a selection mechanism based on supervised learning through a database of chess grandmaster games. The selection mechanism allows that the virtual players which find the largest number of movements proposed by chess grandmasters (according to the database adopted) to pass to the next generation. Also, our evolutionary algorithm has a historic mechanism that allows to recover good virtual players which have temporarily left the evolutionary process.

The evolutionary algorithm is shown in Algorithm 1. 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. In lines 3–15 we carry out the tuning of the weights for the N virtual players during $Gmax$ generations. In line 4, a virtual player's score is incremented by one for each movement of the p positions on the database for which the virtual player did the same action as the human chess master. Line 5 applies the selection mechanism so that only the best $N/2$ virtual players pass to the following generation. Line 6 updates the historical mechanism of virtual players in a way that maintains in an array of the best virtual players which have left the evolutionary process. In lines 7–13 we obtain the second half of virtual players needed. If $100 * rand(0, 1) < P_r$ (where P_r is a control parameter defined by the user), we obtain the virtual player i from the historical mechanism, and if not, we mutate the virtual player $i - N/2$ to obtain the virtual player i. Finally, line 14 increases the generation counter in 1.

The procedure for computing the score of each virtual player is described in Algorithm 2. In lines 1–3, 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 movement 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.

Algorithm 1. EvolutionaryAlgorithm()

```
1:  intializePopulation();
2:  g = 0;
3:  while g < Gmax do
4:    scoreCalculation();
5:    selection();
6:    updateHistoricalMechanism();
7:    for i = N/2 → N − 1 do
8:      if 100 * rand(0, 1) < Pr then
9:        VP[i] ← historicalMechanism();
10:     else
11:       VP[i] ← mutate(VP[i-N/2]);
12:     end if
13:   end for
14:   g++;
15: end while
```

Table 2
Thirty-one runs with $P_r = 0$ for 24 weights.

Number of generations (*Gmax*):	200
Population size:	20
Number of training positions:	4000
P_r:	0

Positions solved by the best VP (%)		Average positions solved by all VPs (%)		Time (min:s)
Generation 0	Generation 200	Generation 0	Generation 200	
17.62	31.77	15.95	31.65	57:23
18.48	32.42	16.38	31.73	57:27
18.48	32.30	13.85	31.75	57:28
18.48	32.55	16.38	31.75	57:31
17.62	31.90	16.38	31.77	57:26
16.80	32.17	14.90	31.92	57:38
17.62	32.22	16.17	32.00	57:35
19.33	32.22	16.17	32.03	57:31
19.33	32.25	17.85	32.05	57:21
18.48	32.38	17.42	32.10	57:29
15.95	32.35	15.32	32.10	57:31
15.12	32.40	13.85	32.15	57:33
20.15	32.65	17.23	32.28	57:23
18.48	32.55	16.58	32.40	57:38
21.00	32.58	17.23	32.55	57:37
21.00	**32.67**	**17.62**	**32.65**	**57:34 (median 25)**
24.35	33.67	21.00	32.67	57:31
16.80	33.20	14.90	32.70	57:27
19.33	33.10	14.90	32.78	57:30
19.33	33.60	17.00	32.80	57:35
22.67	33.22	17.00	32.83	57:27
27.73	33.42	19.33	32.85	57:31
22.67	33.47	19.73	32.92	57:38
19.33	33.30	16.58	33.00	57:32
23.52	33.30	19.10	33.00	57:28
19.33	33.40	16.58	33.10	57:38
21.83	33.42	18.27	33.15	57:26
16.80	33.67	15.32	33.22	57:33
20.15	33.50	17.23	33.28	57:32
21.00	33.67	18.05	33.38	57:21
20.15	33.65	18.27	33.40	57:22

Standard deviation of the fourth column:	0.55
Average run time (min:s):	57:30

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
```

4.1. Initialization

The population of our evolutionary algorithm was initialized with 20 virtual players (10 parents and 10 offspring in subsequent generations). The weight values for these virtual players were randomly generated with a uniform distribution within their allowable bounds. The allowable bounds for each weight is shown in Table 1.

4.2. Mutation

If the condition $100 * rand(0, 1) < P_r$ is true (in Algorithm 1), one offspring is taken from the historical mechanism; otherwise, it is mutated from its corresponding parent (the parent $i - N/2$ is mutated to generate the offspring i, for $i = N/2, \ldots, N - 1$).

The values that were mutated are shown in Table 1.

In our implementation, we adopted Michalewicz's non-uniform mutation operator [20]. The expression to obtain the mutated weight from the previous weight V_k is the following:

$$V'_k = \begin{cases} V_k + \Delta(t, UB - V_k) & \text{if R} = \text{TRUE} \\ V_k - \Delta(t, V_k - LB) & \text{if R} = \text{FALSE} \end{cases} \quad (2)$$

where $[LB, UB]$ is the range of the weight V_k, and $R = flip(0.5)$ (the function $flip(p)$ returns TRUE with a probability p). Michalewicz suggests using:

$$\Delta(t, y) = y * (1 - r^{(1-t/T)^b}) \quad (3)$$

where r is a random real number in the range (0, 1), T is the maximum number of generations and b is a user-defined parameter. In our case, $b = 5$, which is the value recommended by Michalewicz [20].

It should be noted that no crossover operator is employed in our case, since we adopted evolutionary programming (this paradigm models the evolutionary process at the species level and, therefore, it does not incorporate any crossover operator).

5. Experimental results

Our experiments were carried out on a personal computer with a 64-bits architecture, with 3 GB in RAM, having two cores running at 2.8 GHz. The programs were compiled using *g++* in the OpenSuse

Table 3
Thirty-one runs with $P_r = 10$ for 24 weights.

Number of generations (*Gmax*):	200
Population size:	20
Number of training positions:	4000
P_r:	15

Positions solved by the best VP (%)		Average positions solved by all VPs (%)		Time (min:s)
Generation 0	Generation 200	Generation 0	Generation 200	
25.52	45.97	21.75	45.20	59:02
19.73	46.33	16.62	45.25	58:53
20.92	45.67	19.65	45.25	58:58
24.05	45.35	20.33	45.30	58:58
23.23	45.70	19.20	45.33	59:03
21.75	46.33	20.77	45.42	58:48
20.92	45.67	18.52	45.45	59:00
22.02	45.83	19.65	45.70	58:50
19.30	46.17	17.58	45.90	58:59
24.17	46.35	22.15	46.00	58:46
28.62	46.08	24.20	46.00	58:47
22.42	46.20	20.35	46.03	59:00
25.52	46.30	21.83	46.12	59:02
27.02	46.28	24.27	46.12	58:53
18.23	46.25	16.92	46.20	58:59
26.48	**46.35**	**22.38**	**46.28**	**58:57 (median 23)**
26.48	46.38	24.08	46.30	58:59
26.20	47.05	21.50	46.45	58:57
31.20	47.15	24.38	46.45	58:54
26.48	47.03	21.67	46.47	58:44
28.62	47.20	20.80	46.55	58:48
26.08	47.15	20.33	46.65	58:53
20.92	47.33	19.55	46.70	58:58
22.27	47.33	18.85	46.72	59:00
23.38	47.08	21.95	46.78	58:51
24.73	47.03	21.95	46.80	59:01
20.40	46.90	18.70	46.80	59:03
23.77	47.25	21.08	46.80	58:53
24.85	46.90	21.85	46.90	58:48
27.55	47.17	24.20	47.15	59:03
25.67	47.33	21.67	47.20	58:50

Standard deviation of the fourth column:	0.60
Average run time (min:s):	58:55

11.4 operating system. For our virtual players, we used the opening book *Olympiad.abk* included with the graphical user interface *Arena*.[1]

For the training of our experiments we used a database of 4000 games from grandmasters in chess having a rating above 2600 Elo. These experiments used the chess engine described in Section 3.

5.1. Experiment A: tuning the weights

In this case, we tuned the first 24 weights shown in Table 1. We carried out 31 runs using $P_r = 5, 10, \ldots, 35, 40$ and found that $P_r = 10$, 15, 20 produced the highest number of positions properly solved. We carried out the following experiments:

1. 31 runs with $Pr = 0$ and 24 weights (without the historical mechanism). These runs are shown in Table 2.
2. 31 runs with $Pr = 10$ and 24 weights. These runs are shown in Table 3.
3. 31 runs with $Pr = 15$ and 24 weights. These runs are shown in Table 4.
4. 31 runs with $Pr = 20$ and 24 weights. These runs are shown in Table 5.

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

The description of these tables is the following. In the first four rows we describe the number of training generations (200), the population size (20), the number of training positions (4000), and the value of *Pr* employed. Next, the first and second columns describe the percentage of positions solved by the best virtual player at generation 0 and generation 200, respectively. The third and fourth columns describe the average positions solved by all virtual players at generation 0 and at generation 200, respectively. The fifth column describes the execution time for each run. The text in **boldface** represents the run corresponding to the median of the values reported in the fourth column (sorted ascending), with respect to the 31 runs performed. Finally, we also provide the standard deviation of the values from the fourth column, as well as the average execution time for all the runs performed. In these tables we can see that the positions solved for the median run at generation 200 were 32.65%, 46.28%, 42.25% and 40.65% for $Pr = 0$, $Pr = 10$, $Pr = 15$, $Pr = 20$, respectively. In Tables 2 and 3, the weights obtained at generation 200 for the best virtual player were called $VP^{24}_{200,0}$ and $VP^{24}_{200,10}$, respectively.

In Fig. 2 we can see the percentage of positions solved for the best virtual player and the average positions solved by all virtual players for the median run in Table 2. At generation 0, the average positions solved by all virtual players were 17.62%, and 21.00% for the best virtual player, respectively. At generation 200, the average positions solved by all virtual players were 32.65%, and 32.67% for the best virtual player, respectively. Note that this value is

Table 4
Thirty-one runs with $P_r = 15$ for 24 weights.

Number of generations (*Gmax*):	200
Population size:	20
Number of training positions:	4000
P_r:	10

Positions solved by the best VP (%)		Average positions solved by all VPs (%)		Time (min:s)
Generation 0	Generation 200	Generation 0	Generation 200	
19.33	42.25	14.90	41.30	58:27
20.15	42.05	17.23	41.47	58:22
15.95	41.97	15.32	41.53	58:13
19.33	42.15	16.17	41.58	58:13
18.48	41.90	16.38	41.58	58:24
20.15	42.08	17.23	41.58	58:22
18.48	42.22	13.85	41.65	58:16
16.80	41.92	14.90	41.67	58:18
18.48	41.72	16.58	41.67	58:11
18.48	42.17	16.38	41.70	58:16
18.48	41.85	17.42	41.75	58:22
21.00	42.25	17.23	41.78	58:21
19.33	41.97	16.58	41.95	58:12
16.80	42.22	14.90	42.00	58:11
17.62	42.25	15.95	42.08	58:19
21.00	**42.28**	**18.05**	**42.25**	**58:11 (median 11)**
23.52	43.08	19.10	42.28	58:14
22.67	42.75	17.00	42.30	58:23
16.80	42.58	15.32	42.30	58:22
15.12	42.45	13.85	42.35	58:26
22.67	42.47	19.73	42.35	58:25
24.35	43.12	21.00	42.38	58:15
20.15	42.88	18.27	42.40	58:23
17.62	43.33	16.17	42.47	58:09
21.00	42.88	17.62	42.50	58:22
21.83	42.72	18.27	42.50	58:26
17.62	42.90	16.38	42.67	58:23
19.33	43.25	16.58	42.72	58:16
19.33	42.97	17.85	42.75	58:11
27.73	42.80	19.33	42.78	58:18
19.33	43.20	17.00	43.03	58:19

Standard deviation of the fourth column:	0.47
Average run time (min:s):	58:18

competitive with the one reported in [8] after performing the supervised evolution with 32% of positions solved.

In Fig. 3, we can see the percentage of positions solved for the best virtual player and the average positions solved by all virtual players for the median run in Table 3. At generation 0, the average positions solved by all virtual players were 22.38%, and 26.48% for the best virtual player, respectively. At generation 200, the average positions solved by all virtual players were 46.28%, and 46.35% for the best virtual player, respectively. Since we adopted Michalewicz's non-uniform mutation operator [20], the percentage

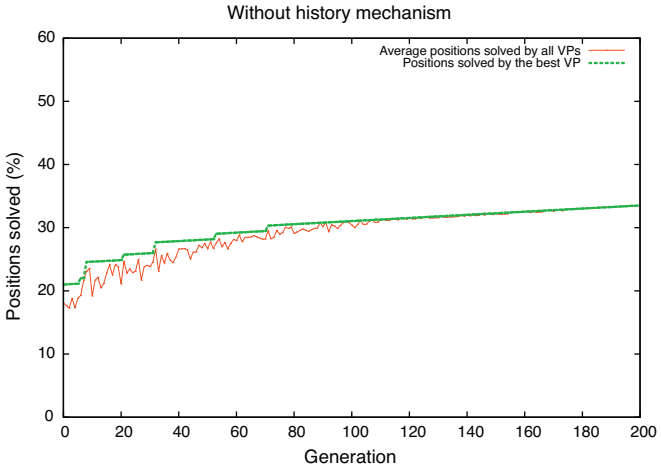

Fig. 2. Evolutionary process without the historical mechanism for $P_r = 0$ with 24 weights. The plot shows the percentage of positions solved for the median run in Table 2 during 200 generations. We considered a total of 4000 positions.

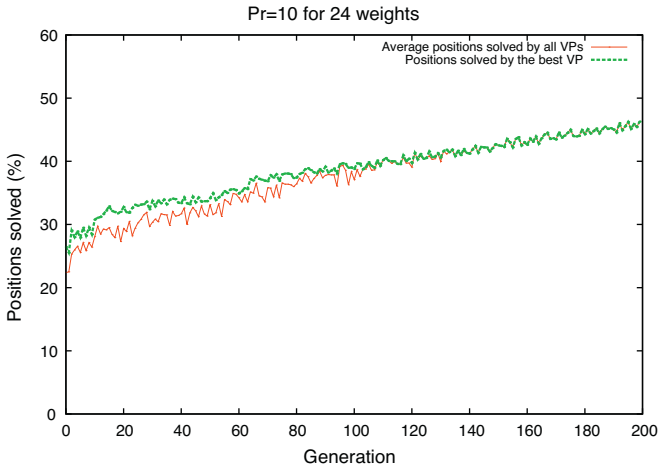

Fig. 3. Evolutionary process for $P_r = 10$ with 24 weights. The plot shows the percentage of positions solved for the median run in Table 3 during 200 generations. We considered a total of 4000 positions.

Table 5
Thirty-one runs with $P_r = 20$ for 24 weights.

Number of generations (*Gmax*):	200
Population size:	20
Number of training positions:	4000
P_r:	20

Positions solved by the best VP (%)		Average positions solved by all VPs (%)		Time (min:s)
Generation 0	Generation 200	Generation 0	Generation 200	
23.12	39.95	20.65	39.58	57:21
21.48	39.65	19.62	39.60	57:11
23.95	40.28	19.40	39.62	57:05
21.48	40.55	18.80	39.70	57:02
21.48	39.72	20.23	39.72	57:21
19.83	40.60	18.38	39.80	57:09
22.30	40.28	20.65	39.80	57:16
23.12	40.42	20.23	39.92	57:05
18.17	40.65	16.73	39.97	57:10
19.83	40.25	18.38	40.15	57:19
23.12	40.50	21.48	40.20	57:05
22.30	40.35	18.38	40.25	57:20
20.65	40.30	18.38	40.25	57:13
22.30	40.62	19.83	40.28	57:21
22.30	40.40	20.45	40.38	57:02
23.95	**40.67**	**21.05**	**40.65**	**57:07 (median 09)**
22.30	40.78	19.62	40.67	57:11
24.77	41.65	21.90	40.75	57:10
20.65	41.47	19.20	40.75	57:13
27.25	41.00	20.65	40.78	57:12
23.12	40.95	19.40	40.80	57:06
23.95	41.67	19.00	40.85	57:02
23.95	41.40	21.27	41.00	57:21
23.12	41.15	20.65	41.03	57:13
27.25	41.53	20.85	41.17	57:12
21.48	41.22	19.20	41.17	57:21
24.77	41.70	22.50	41.20	57:15
19.83	41.53	18.38	41.25	57:07
19.83	41.47	19.20	41.47	57:07
21.48	41.67	20.65	41.53	57:14
24.77	41.75	20.85	41.62	57:02

Standard deviation of the fourth column:	0.63
Average run time (min:s):	57:11

of positions solved was equal for the average of all virtual players and the best virtual player at the end of the evolutionary process (generation 200).

In order to show the scalability of our method, we proceeded to repeat the previous experiments with the 33 weights shown in Table 1. It is worth mentioning that previously, in the paper [27], we already had successfully tuned the weights of the material values and the mobility factor of the pieces.

Again, we carried out 31 runs using $P_r = 5, 10, \ldots, 35, 40$ and we found out that $P_r = 10, 15, 20$ produced the highest number of positions properly solved. We carried out the following experiments:

1. 31 runs with $Pr = 0$ and 33 weights (without the historical mechanism). These runs are shown in Table 6.
2. 31 runs with $Pr = 10$ and 33 weights. These runs are shown in Table 7.
3. 31 runs with $Pr = 15$ and 33 weights. These runs are shown in Table 8.
4. 31 runs with $Pr = 20$ and 33 weights. These runs are shown in Table 9.

The description of these tables is the same as indicated before. In these tables we can see that the positions solved for the median run at generation 200 were 40.78%, 53.08%, 50.10% and 47.55% for $Pr = 0$, $Pr = 10$, $Pr = 15$, $Pr = 20$, respectively. In Tables 6 and 7, the weights obtained at generation 200 for the best virtual player was called $VP^{33}_{200,0}$ and $VP^{33}_{200,10}$, respectively.

In Fig. 4 we can see the percentage of positions solved for the best virtual player and the average positions solved by all virtual players for the median run in Table 6. At generation 0, the average positions solved by all virtual players were 25.42%, and 30.51% for the best virtual player, respectively. At generation 200, the average

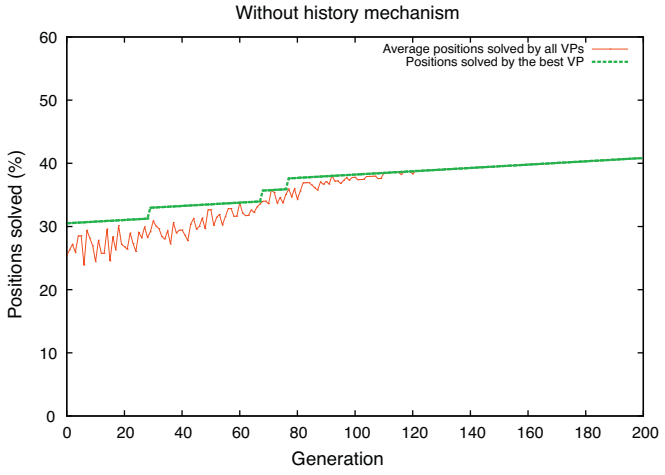

Fig. 4. Evolutionary process without the historical mechanism for $P_r = 0$ with 33 weights. The plot shows the percentage of positions solved for the median run in Table 6 during 200 generations. We considered a total of 4000 positions.

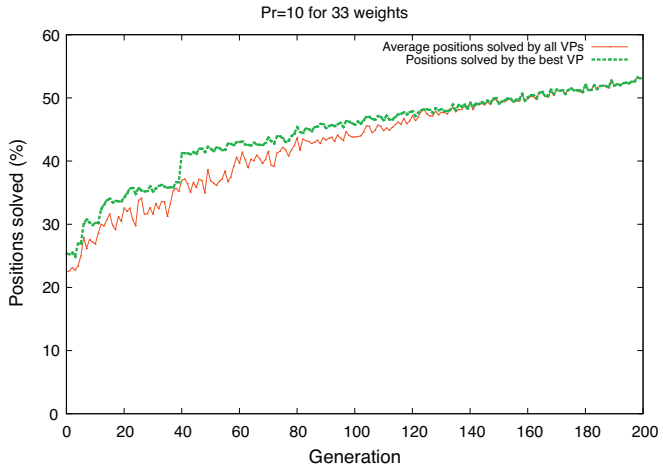

Fig. 5. Evolutionary process for $P_r = 10$ with 33 weights. The plot shows the percentage of positions solved for the median run in Table 7 during 200 generations. We considered a total of 4000 positions.

positions solved by all virtual players were 40.78%, and 40.85% for the best virtual player, respectively.

In Fig. 5 we can see the percentage of positions solved for the best virtual player and the average positions solved by all virtual players for the median run in Table 7. At generation 0, the average positions solved by all virtual players were 22.45%, and 25.38% for the best virtual player, respectively. At generation 200, the average positions solved by all virtual players were 53.08%, and 53.08% for the best virtual player, respectively. Since we adopted Michalewicz's non-uniform mutation operator [20], the percentage of positions solved was equal for the average of all virtual players and the best virtual player at the end of the evolutionary process (generation 200).

At the end of the evolutionary process (generation 200), the percentage of positions solved when using the historical mechanism was larger (53.08%) than those achieved by the version of our algorithm that does not use the historical mechanism (40.78%). Also, the number of positions solved with the historical mechanism is larger than the value reported in [8].

In Table 1, we show the tuning weights for the virtual player $VP^{33}_{200,10}$. In this table we can see that the material values of the pieces are similar to the "theoretical" values known from chess theory [22].

In the experiments of this section, we used a search depth of one ply for our chess engine.

5.2. Experiment B: training cases

In this experiment, we carried out 31 runs with the 33 weights shown in Table 1 for 1000, 2000 and 3000 positions from chess grandmaster games. Table 10 shows in the first, second and third

Table 6
Thirty-one runs with $P_r = 0$ for 33 weights.

Number of generations (*Gmax*):	200
Population size:	20
Number of training positions:	4000
P_r:	0

Positions solved by the best VP (%)		Average positionssolved by all VPs (%)		Time (min:s)
Generation 0	Generation 200	Generation 0	Generation 200	
30.50	40.15	24.58	39.75	63:32
30.50	39.97	23.30	39.75	63:28
28.80	40.10	24.15	39.78	63:22
27.10	39.78	24.15	39.78	63:18
25.40	40.75	23.73	39.83	63:14
28.80	40.47	21.60	39.83	63:18
22.02	39.90	21.17	39.85	63:17
28.80	40.00	22.02	39.95	63:30
25.40	40.60	22.45	40.03	63:22
23.73	40.70	19.90	40.03	63:30
28.80	40.42	25.00	40.08	63:16
25.40	40.20	23.30	40.15	63:19
25.40	40.42	24.15	40.22	63:31
27.10	40.58	22.45	40.42	63:25
30.50	40.60	24.15	40.55	63:31
32.20	**41.12**	**25.83**	**40.78**	**63:24 (median 09)**
30.50	40.83	25.40	40.80	63:25
22.02	41.92	19.90	40.83	63:31
27.10	41.22	22.88	40.85	63:31
28.80	41.58	26.67	40.85	63:33
27.10	41.80	23.73	40.85	63:29
25.40	41.88	22.45	40.85	63:25
23.73	41.65	22.02	40.88	63:21
23.73	41.08	21.17	40.95	63:22
32.20	41.70	27.10	41.03	63:29
28.80	41.08	25.00	41.08	63:30
25.40	41.88	20.33	41.08	63:18
25.40	41.45	21.60	41.08	63:15
25.40	41.25	22.88	41.15	63:28
28.80	41.72	25.40	41.30	63:31
27.10	41.60	24.15	41.55	63:17

Standard deviation of the fourth column:	0.55
Average run time (min:s):	63:25

Table 7
Thirty-one runs with $P_r = 10$ for 33 weights.

Number of generations (*Gmax*):	200
Population size:	20
Number of training positions:	4000
P_r:	10

Positions solved by the best VP (%)		Average positions solved by all VPs (%)		Time (min:s)
Generation 0	Generation 200	Generation 0	Generation 200	
24.80	52.20	21.92	52.00	62:10
25.27	52.12	22.38	52.03	62:15
25.58	52.80	22.65	52.03	62:13
24.90	52.90	21.95	52.12	62:11
25.08	53.05	22.25	52.15	62:11
24.40	52.58	21.58	52.15	62:15
25.17	52.53	22.25	52.20	62:08
24.50	52.33	21.60	52.25	62:27
24.60	52.53	21.75	52.30	62:27
25.17	52.40	22.33	52.33	62:21
25.08	53.00	22.23	52.38	62:25
25.48	53.08	22.55	52.40	62:23
25.27	52.65	22.42	52.53	62:11
25.08	52.85	22.20	52.55	62:10
25.27	52.83	22.40	52.72	62:19
25.38	**53.08**	**22.45**	**53.08**	**62:14 (median 03)**
25.08	53.28	22.20	53.17	62:11
25.08	53.90	22.17	53.22	62:24
25.00	53.40	22.15	53.22	62:22
25.38	54.10	22.45	53.28	62:23
25.27	53.60	22.35	53.30	62:09
25.38	54.17	22.48	53.33	62:26
24.50	53.72	21.65	53.35	62:25
25.17	53.88	22.27	53.35	62:15
24.80	53.88	21.95	53.38	62:20
24.90	53.92	22.02	53.40	62:22
24.80	54.10	21.88	53.42	62:10
25.08	54.17	22.20	53.50	62:23
25.48	53.58	22.55	53.53	62:13
25.08	54.12	22.23	53.53	62:26
25.17	54.10	22.33	53.95	62:18

Standard deviation of the fourth column:	0.59
Average run time (min:s):	62:18

columns, respectively, the training case, the median of the average positions solved by all virtual players and the standard deviation for the 31 runs at generation 200 with $Pr = 10$. The corresponding values for the training case 4000 were taken from the previous experiment. In this table, we can see that as the size of the training case gets smaller, the standard deviation as well as the average positions solved by all the virtual players also gets smaller.

In the experiments of this section, we used a search depth of one ply for our chess engine.

5.3. Experiment C: games versus Chessmaster program

In order to show that the evolutionary process of our method produces virtual players with a higher rating, we carried out 200 games between $VP^{24}_{200,10}$ versus $VP^{24}_{200,0}$, $VP^{24}_{200,10}$ versus $VP^{24}_{0,0}$ and $VP^{24}_{200,0}$ versus $VP^{24}_{0,0}$ (each virtual player played 100 games with white pieces and 100 games with black pieces). The virtual player $VP^{24}_{200,10}$'s losses, draws, and wins were 0, 97, and 103 games, respectively, when playing versus the virtual player $VP^{24}_{200,0}$. The virtual player $VP^{24}_{200,10}$'s losses, draws, and wins were 0, 5, and 195 games, respectively, when playing versus the virtual player $VP^{24}_{0,0}$. Finally, the virtual player $VP^{24}_{200,0}$'s losses, draws, and wins were 0, 14, and 186 games, respectively, when playing versus the virtual player $VP^{24}_{0,0}$.

Based on these played games, we used the Bayeselo tool[2] to estimate the ratings of players using a minorization–maximization algorithm [17]. The obtained ratings are shown in Table 11. In this table, the column "Rank" assigns a rank to the strength of the players, the column "Name" gives the player's name, the column "Elo" gives the ELO rating, the column "Games" gives the total number of games, the column "Score" gives the percentage of points obtained, the column "Oppo." gives the average rating of opponents, and the column "Draws" gives the percentage of draws obtained. In this table, we can see that the rating for the virtual player $VP^{24}_{200,10}$ was 2341, the rating for the virtual player $VP^{24}_{200,0}$ was 2186, and the rating for the virtual player $VP^{24}_{0,0}$ was 1685, representing an increase of 656 rating points between the evolved virtual player with the historical mechanism, and the virtual player without using evolution.

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 15, we can see that the strength of the virtual player $VP^{24}_{200,10}$ (2341 rating points) is at the level of a chess master, the strength of the virtual player $VP^{24}_{200,0}$ (2186 rating points) is at the level of a chess expert, and the strength of the virtual player $VP^{24}_{0,0}$ (1685 rating points) is at the level of a class B chess player.

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

Table 8
Thirty-one runs with $P_r = 15$ for 33 weights.

Number of generations (*Gmax*):	200
Population size:	20
Number of training positions:	4000
P_r:	15

Positions solved by the best VP (%)		Average positions solved by all VPs (%)		Time (min:s)
Generation 0	Generation 20	Generation 0	Generation 200	
18.00	50.00	17.00	49.12	63:15
22.00	50.05	16.00	49.20	63:13
24.00	49.78	20.00	49.22	62:59
24.00	50.20	15.00	49.25	63:04
26.00	49.70	17.50	49.28	63:05
18.00	49.60	14.50	49.30	63:13
26.00	49.53	24.00	49.30	63:05
18.00	49.88	16.00	49.35	63:03
22.00	49.67	14.50	49.40	63:11
20.00	49.75	16.00	49.47	63:13
24.00	49.58	17.00	49.55	63:00
30.00	49.62	23.50	49.62	63:10
22.00	50.10	18.00	49.65	63:15
30.00	49.95	22.62	49.72	63:06
20.00	50.05	14.00	49.83	63:01
26.00	**50.20**	**21.00**	**50.10**	**63:14 (median 29)**
26.00	50.67	17.00	50.15	63:14
22.00	50.28	16.00	50.17	63:13
22.00	50.88	18.00	50.22	63:00
24.00	51.05	19.50	50.22	63:10
28.00	50.65	19.00	50.22	63:07
24.00	51.10	18.00	50.28	63:06
24.00	51.05	17.50	50.30	62:58
20.00	50.47	16.50	50.33	63:05
18.00	50.72	14.00	50.35	63:03
22.00	50.60	19.00	50.47	63:05
24.00	51.22	19.00	50.50	62:57
22.00	50.70	19.50	50.60	63:06
20.00	51.20	18.00	50.62	63:12
28.00	51.12	19.50	50.67	63:03
18.00	50.95	15.50	50.95	63:00

Standard deviation of the fourth column:	0.54
Average run time (min:s):	63:07

Also, we carried out 200 games between $VP^{33}_{200,10}$ versus $VP^{33}_{200,0}$, $VP^{33}_{200,10}$ versus $VP^{33}_{0,0}$ and $VP^{33}_{200,0}$ versus $VP^{33}_{0,0}$ (as before, each virtual player played 100 games with white pieces and 100 games with black pieces). The virtual player $VP^{33}_{200,10}$'s losses, draws, and wins were 0, 96, and 104 games, respectively, when playing versus the virtual player $VP^{33}_{200,0}$. The virtual player $VP^{33}_{200,10}$'s losses, draws, and wins were 0, 1, and 199 games, respectively, when playing versus the virtual player $VP^{33}_{0,0}$. Finally, the virtual player $VP^{33}_{200,0}$'s losses, draws, and wins were 0, 2, and 198 games, respectively, when playing versus the virtual player $VP^{33}_{0,0}$. Again, we used the Bayeselo tool to estimate the ratings of players using a minorization–maximization algorithm [17]. The obtained ratings are shown in Table 12. In this table, we can see that the rating for the virtual player $VP^{33}_{200,10}$ was 2402, the rating for the virtual player $VP^{33}_{200,0}$ was 2245, and the rating for the virtual player $VP^{33}_{0,0}$ was 1509, representing an increase of 893 rating points between the evolved virtual player with the historical mechanism, and the virtual player without using evolution.

Finally, we carried out an additional experiment between the virtual players and Chessmaster$_{2400}$, which is a version of the popular chess program *Chessmaster* (grandmaster edition) which plays at 2400 rating points. Specifically, we played 200 games among Chessmaster$_{2400}$ and each of the virtual players $VP^{33}_{200,10}$, $VP^{24}_{200,10}$, $VP^{33}_{200,0}$, and $VP^{24}_{200,0}$ (the number of games with white pieces and black pieces was the same for the virtual players and the program *Chessmaster*; for example, *Chessmaster* played 400 games with white pieces and 400 with black pieces). The results are shown in Fig. 6. In this figure, we can see that Chessmaster$_{2400}$'s losses, draws, and wins were 14, 169, and 17, respectively, when playing versus the virtual player $VP^{33}_{200,10}$ (denoted as H1 in this Figure). Also, Chessmaster$_{2400}$'s losses, draws, and wins were 10, 137, and 53, respectively, when playing versus the virtual player $VP^{24}_{200,10}$ (denoted as H2 in this Figure), respectively, and so on.

Based on these played games, we used again the Bayeselo tool to estimate the ratings of the virtual players and Chessmaster$_{2400}$. The obtained ratings are shown in Table 13. In this table we can see that the rating for the virtual players $VP^{33}_{200,10}$, $VP^{24}_{200,10}$, $VP^{33}_{200,0}$, and $VP^{24}_{200,0}$ were 2397, 2344, 2249, and 2194 rating points, respectively.

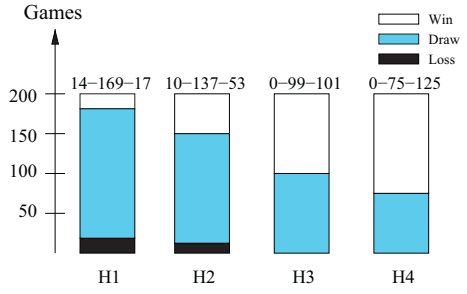

Fig. 6. Histogram of losses, draws and wins for Chessmaster$_{2400}$ against $VP^{33}_{200,10}$ (H1), $VP^{24}_{200,10}$ (H2), $VP^{33}_{200,0}$ (H3), and $VP^{24}_{200,0}$ (H4).

Table 9
Thirty-one runs with $P_r = 20$ for 33 weights.

Number of generations (*Gmax*):	200
Population size:	20
Number of training positions:	4000
P_r:	20

Positions solved by the best VP (%)		Average positions solved by all VPs (%)		Time (min:s)
Generation 0	Generation 200	Generation 0	Generation 200	
27.10	46.85	20.75	46.58	62:48
25.40	47.17	22.02	46.58	62:58
27.10	46.97	24.15	46.62	63:03
23.73	47.35	22.02	46.62	62:56
23.73	47.53	24.58	46.62	62:54
22.02	47.00	20.75	46.67	62:44
25.40	47.53	18.62	46.78	62:54
20.33	47.40	19.05	46.80	63:03
28.80	47.53	22.45	46.83	63:03
23.73	46.95	21.17	46.83	62:59
23.73	46.90	22.45	46.88	63:00
22.02	47.28	20.33	46.92	62:59
27.10	47.03	21.17	46.97	63:00
27.10	47.58	22.45	46.97	62:56
27.10	47.20	24.15	47.12	62:45
22.02	**47.62**	**19.90**	**47.55**	**63:03 (median 12)**
22.02	48.30	17.35	47.62	62:44
25.40	48.35	19.90	47.62	62:48
23.73	47.88	23.30	47.78	62:52
27.10	47.92	22.88	47.78	63:03
25.40	48.10	22.45	47.85	62:57
23.73	48.17	19.48	47.88	62:59
25.40	48.42	23.30	47.90	62:51
23.73	48.35	21.60	47.92	62:45
22.02	48.25	19.90	48.10	62:48
25.40	48.45	23.30	48.10	62:48
27.10	48.33	22.45	48.15	62:46
27.10	48.38	22.02	48.25	62:46
25.40	48.50	22.45	48.33	63:01
22.02	48.45	18.20	48.35	62:58
23.73	48.55	21.17	48.55	62:45

Standard deviation of the fourth column:	0.65
Average run time (min:s):	62:54

Table 10
It shows the median of the average positions solved by all virtual players and the standard deviation for 31 runs at generation 200 and $Pr = 10$ with different training case.

Training case	Average positions solved by all virtual players	Standard deviation	Average evaluation time
1000	50.87%	0.44	14:37
2000	51.07%	0.47	31:12
3000	52.02%	0.51	45:37
4000	53.08%	0.59	62:18

Table 11
Ratings for $VP^{24}_{200,10}$, $VP^{24}_{200,0}$ and $VP^{24}_{0,0}$.

Rank	Name	Elo	Games	Score (%)	Oppo.	Draws (%)
1	$VP^{24}_{200,10}$	2341	400	87%	1936	26%
2	$VP^{24}_{200,0}$	2186	400	60%	2014	28%
3	$VP^{24}_{0,0}$	1685	400	2%	2264	5%

In the experiments of this section, we used a search depth of six plies for our chess engine, and in the opening phase we used the database *Olympiad.abk* included with the graphical user interface *Arena*.[3]

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

Table 12
Ratings for $VP^{33}_{200,10}$, $VP^{33}_{200,0}$ and $VP^{33}_{0,0}$.

Rank	Name	Elo	Games	Score (%)	Oppo.	Draws (%)
1	$VP^{33}_{200,10}$	2402	400	88%	1877	24%
2	$VP^{33}_{200,0}$	2245	400	62%	1955	25%
3	$VP^{33}_{0,0}$	1509	400	0%	2324	1%

5.4. Experiment D: hybrid approach

This experiment is divided in two stages. The first stage corresponds to the experiment described in Section 5.1, and consisted of adjusting the weights of the 20 virtual players through chess grandmaster games (this is known as supervised learning). In the second stage, we applied an evolutionary algorithm (also based on evolutionary programming [13]) to perform a tournament between virtual players (this is known as unsupervised learning or co-evolution).

Table 13
Ratings for $Chessmaster_{2400}$ and the virtual players in generation 200.

Rank	Name	Elo	Games	Score (%)	Oppo.	Draws (%)
1	$Chessmaster_{2400}$	2401	800	67%	2296	60%
2	$VP^{33}_{200,10}$	2397	200	49%	2401	85%
3	$VP^{24}_{200,10}$	2344	200	39%	2401	69%
4	$VP^{33}_{200,0}$	2249	200	25%	2401	50%
5	$VP^{24}_{200,0}$	2194	200	19%	2401	38%

Table 14
Ratings for $VP^{33}_{200,10}$ and VP^{33}_{50}.

Rank	Name	Elo	Games	Score (%)	Oppo.	Draws (%)
1	VP^{33}_{50}	2442	200	57%	2404	61%
2	$VP^{33}_{200,10}$	2404	200	43%	2442	61%

Specifically, we carried out a tournament among $n = 20$ virtual players. Initially, the weights for this n virtual players were taken from the median run in Table 7. Afterwards, each virtual player was allowed to play $n/2$ games with randomly chosen opponents. The side (either black or white) was chosen at random. Games were executed until one of the virtual players received checkmate or until a draw condition arose. Depending on the outcome of the game, a virtual player obtained one point, half a point or zero points for a win, draw or loss, respectively. Draw conditions were given by the rule of 50 moves (after a pawn's move there are 50 moves to give a checkmate to the opponent), by the third repetition of the same position and by the lack of victory conditions (e.g., a king and a bishop versus a king).

After finishing the tournament, the selection mechanism chooses the $n/2$ virtual players having the highest number of points, and subsequently these virtual players are mutated to generate the remaining $n/2$ virtual players. Finally, the evolutionary algorithm continues running for 50 generations.

As we saw in Section 5.1, the best virtual player obtained in the first stage was called $VP^{33}_{200,10}$, and the best virtual player obtained in the second stage was called VP^{33}_{50}.

Finally, we carried out 200 games between $VP^{33}_{200,10}$ versus VP^{33}_{50}, (again each virtual player played 100 games with white pieces and 100 games with black pieces). The virtual player $VP^{33}_{200,10}$'s losses, draws, and wins were 53, 122, and 25 games, respectively, when playing versus the virtual player VP^{33}_{50}. Again, we used the Bayeselo tool to estimate the ratings of the virtual players. The obtained ratings are shown in Table 14.

In this table, we can see that the rating for the virtual player $VP^{33}_{200,10}$ was 2404, and that the rating for the virtual player VP^{33}_{50} was 2442, representing an increase of 38 rating points between the virtual player obtained with supervised learning, and the virtual player obtained with unsupervised learning.

In the experiments of this section, we used a search depth of six plies for our chess engine, and in the opening phase we used the database *Olympiad.abk*.

5.5. Validation

With the completion of the supervised learning, we used an additional 4000 positions for testing purposes. Specifically, we let the virtual player $VP^{33}_{200,0}$ perform a 1-ply search on each of these positions, and the percentage of correctly solved positions was 39.7%. Also, we allowed the virtual player $VP^{33}_{200,10}$ to perform a 1-ply search on each of these positions, and the percentage of correctly solved positions was 52.1%. This indicates that the first 4000 positions used for training cover most of the types of positions that can arise.

Table 15
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

6. Conclusions and future work

Most of the works that make use of evolutionary algorithms to tune the weights of a chess engine adopt co-evolution. In these methods, the virtual players hold tournaments among them, and the virtual players which obtain a larger number of victories acquire the right to pass to the next generation. Our proposed approach uses supervised learning to perform the tuning of weights of a chess engine through a database of chess grandmaster games. The idea of the selection mechanism of our evolutionary algorithm is to favor virtual players that are able to "visualize" (or match) more movements from those registered in a database of chess grandmaster games. With our proposal, and after 200 generations, our virtual players can solve 40.78% of the positions of chess grandmaster games. Additionally, our evolutionary algorithm employs a historical mechanism that allows it to successfully solve 53.08% of the positions of chess grandmaster games. Note that this value is higher than the value of 32.4% reported in the work of David-Tabibi et al. [8], who also used supervised learning with a database of chess grandmaster games, unsupervised learned (through co-evolution) and a genetic algorithm. It is noteworthy that our work uses evolutionary programming, and David-Tabibi's work uses a genetic algorithm. We believe that the number of positions solved by our method is satisfactory considering that this was achieved adopting only a depth of 1 ply in the search tree.

In the games held between our virtual players and *Chessmaster* (playing at a rating of 2400) we found that the best evolved virtual player without the historical mechanism played at 2249 rating points, and the best evolved virtual player with the historical mechanism played at 2397 rating points. Thus, we conclude that the rating of our chess engine is competitive with that of the version of *Chessmaster* adopted in our study.

Additionally, we used unsupervised learning (through a tournament held among virtual players) to improve the rating obtained with supervised learning. In this case, we obtained an improvement of 38 rating points (from 2404 to 2442 rating points).

We note that the standard deviation increases as we increase the size of the test cases. Also, our results indicate that the values of the chess pieces obtained by our proposed approach closely match the known values from chess theory.

As part of our future work, and with the idea of creating a chess program that is able to play better, we plan to add and tune more weights in our evaluation function. We also intend to add extensions to our program (e.g., passed pawn extensions, mate-threat, among others), and we plan to use bitbases or tablebases in the final phase of the game, aiming to increase the rating of our chess engine.

Acknowledgements

We thank the reviewers for the valuable comments which greatly helped us to improve the contents of this paper.

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 that calculates the relative strength of players in games with two opponents, and was

created by the mathematician Arpad Elo. In Table 15 we show the classification of the USCF (United States Chess Federation).

References

[1] D.F. Beal, M.C. Smith, Learning piece values using temporal differences, Journal of The International Computer Chess Association 20 (September (3)) (1997) 147–151.

[2] D.F. Beal, M.C. Smith, Learning piece-square values using temporal differences, Journal of The International Computer Chess Association 22 (December (4)) (1999) 223–235.

[3] B. Bošković, J. Brest, A. Zamuda, S. Greiner, V. Žumer, History mechanism supported differential evolution for chess evaluation function tuning, Soft Computing – A Fusion of Foundations, Methodologies and Applications, in press.

[4] B. Bošković, S. Greiner, J. Brest, V. Žumer, A differential evolution for the tuning of a chess evaluation function, in: 2006 IEEE Congress on Evolutionary Computation, Vancouver, BC, Canada, July 16–21, IEEE Press, 2006, pp. 1851–1856.

[5] B. Bošković, S. Greiner, J. Brest, A. Zamuda, V. Žumer, An adaptive differential evolution algorithm with opposition-based mechanisms, applied to the tuning of a chess program, in: U. Chakraborty (Ed.), Advances in Differential Evolution, 2008, pp. 287–298, Springer, Studies in Computational Intelligence, vol. 143, Heidelberg, Germany.

[6] D. Breuker, J.W.H.M. Uiterwijk, H.J.V.D. Herik, Information in transposition tables, Advances in Computer Chess 8 (1997) 199–211.

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

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

[9] T. Ellman, Explanation-based learning: a survey of programs and perspectives, ACM Computing Surveys 21 (June (2)) (1989) 163–221.

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

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

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

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

[14] A. Hauptman, Gp-endchess: using genetic programming to evolve chess endgame players, in: Proceedings of 8th European Conference on Genetic Programming (EuroGP2005), Springer, 2005, pp. 120–131.

[15] A. Hauptman, 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, Berlin, Heidelberg, Springer-Verlag, 2007, pp. 78–89.

[16] F. Hsu, T. Anantharaman, M. Campbell, A. Nowatzyk, Deep thought, in: Computers, Chess and Cognition, Springer, Berlin, 1990, pp. 55–78 (Chapter 5).

[17] R. Hunter, MM algorithms for generalized Bradley-Terry models, The Annals of Statistics 32 (2004) 2004.

[18] G. Kendall, G. Whitwell, An evolutionary approach for the tuning of a chess evaluation function using population dynamics, in: Proceedings of the 2001 Congress on Evolutionary Computation CEC2001, vol. 2, May, IEEE Press, 2001, pp. 995–1002.

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

[20] Z. Michalewicz, Genetic Algorithms +Data Structures = Evolution Programs, 2nd ed., Springer-Verlag, 1996.

[21] H. Nasreddine, H. Poh, 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), IEEE Press, 2006, pp. 1–6.

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

[23] R.S. Sutton, A.G. Barto, A temporal-difference model of classical conditioning, in: Ninth Annual Conference of the Cognitive Science Society, Hillsdale, New Jersey, USA, July, Lawrence Erlbaum Associates, Inc., 1987, pp. 355–378.

[24] S. Thrun, Learning to play the game of chess, in: G. Tesauro, D. Touretzky, T. Leen (Eds.), Advances in Neural Information Processing Systems (NIPS), vol. 7, MIT Press, Cambridge, MA, 1995, pp. 1069–1076.

[25] A. Turing, Digital Computers Applied to Games, of Faster than Thought, Pitman, 1953, pp. 286–310 (Chapter 25).

[26] E. Vázquez-Fernández, C.A.C. Coello, F.D.S. Troncoso, An adaptive evolutionary algorithm based on typical chess problems for tuning a chess evaluation function, in: Proceedings of the 13th Annual Conference Companion on Genetic and Evolutionary Computation, GECCO'11, Dublin, Ireland, July 12–16, ACM, 2011, pp. 39–40.

[27] E. Vázquez-Fernández, C.A.C. Coello, 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.