An Evolutionary 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*—This paper proposes a method for tuning the weights of the evaluation function of a chess program whose search engine is based on evolutionary programming. In our proposed approach, each individual in the population of the evolutionary algorithm represents a virtual player with specific weights of its evaluation function. This differs from most of the previous approaches reported in the literature, in which normally a tournament between virtual players is held, and the final result (win, loss or draw) is used to decide which players will pass to the following generation. The selection mechanism of our proposed algorithm uses games from chess grandmasters to decide which virtual player will pass to the following generation. Our results indicate that the weight values obtained by our approach are similar to the values known from chess theory. Additionally, the standard deviation from the different runs performed, are lower than those reported by authors of previous related approaches.**

Index terms: evolutionary algorithms, chess, computational intelligence in games.

I. Introduction

For the past 61 years, chess has been a very active area of research in artificial intelligence. In 1950, Claude Shannon [20] published the first paper on a computer chess problem. In that paper, Shannon distinguished two strategies: the first that looks at all continuations, and the second, that cuts off certain continuations. In 1953, Alan Turing [23] provided the first description of how to design a computer program capable of playing a full game of chess. In 1975, Donald Knuth [17] provided a detailed analysis of the alpha-beta pruning algorithm, which uses a tree to represent the movements of a game with two adversaries. This is the most commonly adopted algorithm within chess-playing programs, and it has the advantage of refraining from evaluating some nodes when unnecessary (i.e., it uses a pruning technique). The development of these traditional, but computationally expensive algorithms for playing chess reached a high point with the defeat of Garry Kasparov, who was then the World Chess Champion, by IBM's chess computer Deep Blue in 1997 [7]. This computer had a processing speed of about 200 million positions per second.

The main components of a chess engine are: the move generator, the search function and the evaluation function. As its name indicates, the *move generator* generates all possible movements from a certain (given) position, the *search function* (mainly when using the alpha-beta pruning algorithm) finds the best variants from a certain (given) position on the board, and the *evaluation function* is used to heuristically determine the relative value of a position which is then employed in the search algorithm. Additionally, the chess engine can be supplemented with quiescent search and hash tables. *Quiescent search* [1] allows us to stabilize the positions, and *hash tables* [6] allow us to store positions so that they don't have to be looked for again.

The evaluation function is the most important part of a chess engine. The evaluation function contains arithmetic expressions and weights which encode specific knowledge that constitutes a very valuable source of information for the search engine. If the weights used in the evaluation function are improved, then the chess engine will be better (i.e., it will play better). 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.

Most of the previous related work that has been reported in the specialized literature [9], [10], [11], [4], [5], [16], [19] adopts tournaments between virtual players from which the final result of each game (win, loss or draw) is used for deciding which players will pass to the following generation.

In the work reported in this paper, we carried out the automatic tuning of the weights of our evaluation function using an evolutionary algorithm. The selection mechanism of the proposal presented here uses games from chess grandmasters to decide which virtual player will pass to the following generation. Our results indicate that the weight values obtained by our proposed approach match the values that are known from chess theory.

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

II. Previous Related Work

The first program which learned to play chess from final outcomes was NeuroChess [22] and its evaluation function was represented by neural networks. This work also included both temporal difference learning [21] and explanation-based learning [8]. The Deep Thought (later called Deep Blue) team tuned the weights of their evaluation function using a database of grandmaster-level games [14]. In this case, the authors did not incorporate any sort of evolutionary algorithm into their chess engine. Instead, they adopted two approaches: the first, was based on a standard hillclimbing algorithm and simulated annealing to avoid getting stuck at local optima, and the second was based on the system of equalities formulation [14]. Beal and Smith [2] used temporal-difference learning to determine the piece values of a chess program. In a further paper, they incorporated piece-square values into their work [3]. Gomboc [13] proposed an empirical gradient method to optimize the weights of a chess evaluation function. The resulting weights turned out to be very similar to a set of hand-tuned weights.

Evolutionary algorithms have also been used before for tuning the weights of the evaluation function of a chess engine. Kendall and Whitwell [16] showed how the outcome of the game (win, loss or draw) can be used to adjust the weights of a chess engine evaluation function. Nasreddine et al. [19] proposed a new strategy called "dynamic boundary strategy" in which the boundaries of the interval of each parameter are dynamic. Fogel et al. [9] presented an evolutionary algorithm in a computer chess program to learn chess by playing games against itself. They improved the rating of its program in 400 rating points and tuned the material values of the pieces, their positional values, and the weights of three neural networks. In a second work, Fogel et al. [10] evolved their program during 7462 generations, reaching a rating of 2650. The resultant program was called *Blondie25*. In a third work Fogel et al. [11], incorporated to *Blondie25* a heuristic for time management, achieved a rating of 2635 points against the program $Fritz8.0$ who was rated $\#5$ in the world. *Blondie25* was also the first machine learning based chess program able to defeat a human chess master. Bošković et al. [4] presented a differential evolution algorithm for tuning the chess material values and the mobility factor of a chess engine. The weights obtained with this method, matched the values known from chess theory. In a further work, Bošković et al. [5] also used differential evolution to adjust the weights of the evaluation function of a chess program. In this work, they employed adaptation and opposition-based optimization mechanisms. This work presented a better convergence than previous related work.

It is important to emphasize that in all of the previous work reviewed in this section, in which evolutionary algorithms were adopted in some way, the proposed approaches use the final results of a game (win, loss or draw) to decide which individuals will pass to the following generation. None of them uses information from chessmaster's games to make this decision, as we do in this paper.

III. Our Chess Engine

In order to conduct our experiments, we created a chess program that could incorporate a variety of learning strategies to improve the rating of our search engine. In our program, in order to select a movement at each player's turn, a minimax tree is generated and the alpha-beta algorithm with pruning [17] is applied to a fixed depth of 1 (as recommended in [14]). Quiescence is used to extend the search tree to steady positions in which material exchanges cannot influence the resulting evaluation of the position. The program also adopts hash tables and iterative deepening [6]. During the evolutionary process, our chess program uses the same type of evaluation function adopted by Bošković et al. in [4]:

$$eval = X_m(M_{white} - M_{black}) + \sum_{y=0}^{5} X_i(N_{y,white} - N_{y,black}) \quad (1)$$

In this equation, X_i represents the weights for all pieces except for the king. The king's weight was not taken into account because in eq. (1), its associated term is zero (there is always a king for each side on the board). M_{white} represents the number of available moves (mobility) for the white pieces, and M_{black} represents the mobility for the black pieces. X_m is the mobility weight. $N_{y,white}$ and $N_{y,black}$ are the number of y pieces for the white or the black pieces, respectively. y can denote a queen, rook, bishop or knight. The weight for the pawn is always 100.

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 an evolutionary algorithm (in our case, we adopted evolutionary programming [12]), so that they closely match the values derived from chess theory. This sort of evaluation function is relatively simple, but still provides a reasonably good search strategy for our chess engine. It is worth adding that the training of our search engine was conducted using a database of games from chess grandmasters.

IV. Our Proposed Approach

As indicated before, our proposed approach is based on an evolutionary algorithm which has a selection mechanism based on a database of chess grandmasters games. The idea is that the weights adopted in our evaluation function are such that the movement performed is equal to the one that was performed by a human chess master in a particular game from the database. This similarity is used to decide which virtual

player (individuals in the population) will pass to the following generation.

Fig. 1 shows the flow chart of our proposed evolutionary algorithm for tuning the weights of the evaluation function given in eq. (1).

At first, the weights of N virtual players are initialized with random values within their corresponding boundaries. In our experiments, N is equal to ten. Subsequently, a virtual player's score is incremented in one for each movement of the P games on the database for which the virtual player did the same action as the human chess master.

The value of the parameter P is provided by the user and refers to the number of games that will be (randomly) chosen from the database to calculate the score of a virtual player for a generation. The selection procedure chooses the $N/2$ virtual players that achieved the highest score. These virtual players are allowed to pass to the next generation and, consequently, will be allowed to generate offspring using mutation, in order to give rise to the new population of N virtual players. In our experiments, this procedure is repeated during 50 generations.

The procedure for computing the score of each virtual player is described in Algorithm IV.1. Line 1 gets the set S which consists of P games chosen at random from the database. Parameter P ranges from 1 to the number of games available in the database (in our case 312). In lines 2 to 4, we establish the score counter to zero for each virtual player. Line 5 chooses d training games from S. Line 6 sets the starting position of the game d. Line 7 chooses the next movement m from the game d. Finally, each virtual player calculates his next move n, and if this movement matches the movement m, this virtual player increases his score in 1.

Algorithm IV.1: scoreCalculation()

```
1  S = chooseGames(P);
2  for each virtual player i do
3  |   score[i] = 0;
4  end
5  for each game d in S do
6  |   setPosition(d);
7  |   for each movement m in game d do
8  |   |   for each virtual player i do
9  |   |   |   n = nextMovement(i);
10 |   |   |   if m == n then
11 |   |   |   |   score[i]++;
12 |   |   |   end
13 |   |   end
14 |   end
15 end
```

A. Initialization

The population of our evolutionary algorithm was initialized with 10 virtual players (5 parents and 5 offspring in subsequent generations). The weight values for these virtual players were random values generated with a uniform distribution within the allowable bounds. The allowable bound for each piece and for mobility weight are described in Section V.

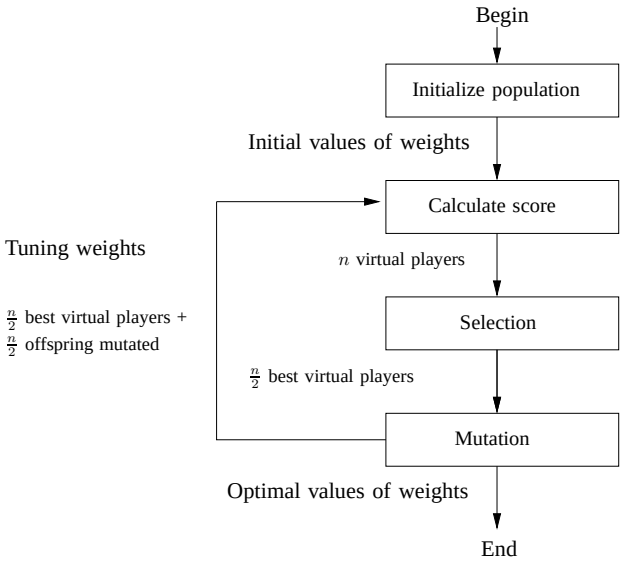

Fig. 1. Flowchart of our proposed evolutionary algorithm.

B. Mutation

One offspring was created by mutating all weights from each surviving parent with a probability of 90% (we carried out several runs using mutation rates of 80%, 85%, 90%, 95% and 100%, and found that 90% produced the best convergence and standard deviation values). The values that were mutated were the following:

- The material values of the knight, bishop, rook and queen. The pawn's weight was fixed at 100 points.
- The mobility of the position.

Our implementation adopted Michalewicz's non-uniform mutation operator [18]. In this operator, the mutated weight $V_k^{'}$ (obtained from the previous weight V_k) is obtained with the following expression:

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

where the weight V_k is within the range $[LB, UB]$ and $R = flip(0.5)$. The function $flip(p)$ simulates the tossing of a coin and 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 between 0 and 1. T is the maximum number of generations and b is a user-defined parameter. In our case, $b = 2$.

Since we adopted evolutionary programming, no crossover operator is employed in our case.

C. Database of Games

The database that we adopted consists of 312 games taken from the Linares super tournament in its editions 1999, 2001, 2002, 2003, 2004, 2005, 2008 and 2010. These games can

be downloaded from: *http://www.chessbase.com/*. Clearly, the database can be expanded so that a more robust tuning of weights can be performed, but this is not particularly relevant at this point, since our main aim here is to present some proof-of-principle results of our proposed methodology.

V. Experimental Results

A. Tuning weights

In our experiments, we tuned the weights of the pieces and their mobility as shown in eq. (1). N (our population size) was set to 10, and the number of training games P was set to 6. Initialization took place using randomly generated values within the vicinity of their "theoretical" values (± 200 points). The "theoretical" values of the pieces are: 300, 330, 500 and 900 for the knight, bishop, rook and queen, respectively. The "theoretical" value of the mobility weight is 10, and its bounds are $[0, 300]$. The "theoretical" values are obtained from [20]. If any of the parameters fell outside its allowable bounds after mutation, it was set to its maximum or minimum allowable value, depending on the boundary exceeded. 30 runs were carried out under these conditions, and in all of them, the "theoretical" values were reached for all pieces.

In order to visualize better the convergence process, we carried out an additional run (number 31) in which the material values were generated within the range $[400, 500]$ and the mobility weight was set in the interval $[0, 300]$. For this run, the average weight values and their standard deviations are shown in Table I.

TABLE I
AVERAGE WEIGHT VALUES AND THEIR STANDARD DEVIATIONS FOR RUN NUMBER 31 (GENERATION 0)

Weight	Value	Standard deviation
X_{pawn}	100.00	0.00
X_{knight}	499.42	34.98
X_{bishop}	464.60	88.67
X_{rook}	469.85	122.75
X_{queen}	437.57	85.90
$X_{mobility}$	97.19	173.67

At the end of run 31, and after 50 generations, the average weight values and their standard deviations are shown in Table II.

TABLE II
AVERAGE WEIGHT VALUES AND THEIR STANDARD DEVIATIONS FOR RUN NUMBER 31 (GENERATION 50)

Weight	Value	Standard deviation
X_{pawn}	100.00	0.00
X_{knight}	310.89	0.22
X_{bishop}	325.32	0.45
X_{rook}	514.92	1.26
X_{queen}	841.61	2.62
$X_{mobility}$	5.62	1.34

The average weight values and their standard deviations for 50 generations are shown in Figs. 2 and 3, respectively.

From the obtained results, we can see that the tuning process after 50 generations resulted in standard deviation values which are lower than those reported by [4] and [19]. This indicates that our proposed approach is more robust.

The computational time required by our proposed approach to run during 50 generations was 3 minutes with 34 seconds under the operating system openSuse, using a PC with a 64 bits architecture, having two cores running at 2.8 Ghz. Unfortunately, most of the references that we consulted do not report any CPU times to have an idea of the efficiency of our approach. The only reference in which we found such information is [9], in which Fogel reported using a 2.2-Ghz Celeron PC with 128 MB of RAM. His program required 36 hours for executing 50 generations. However, it is important to indicate that he optimized many more weights that our approach (namely, the weights of three neural networks, the weights of the positional values, etc.) and adopted a search depth of 4 ply. Thus, this execution time is not comparable with ours and is provided here just as a reference.

It is also worth indicating that the CPU time required by our proposed approach depends on the number of games that are randomly chosen from the database to compute the score of a virtual player during a generation of our evolutionary algorithm. In our case, we adopted $P = 6$.

B. Additional Games

We also performed an additional experiment. We carried out 100 games in which the first virtual player adopted the average weights from generation 0 of our evolutionary algorithm and the second adopted the average weights from generation 50. The scores achieved by them were 84 of the second player (who used the weights from generation 50) versus 16 from the first player. In Appendix A, we show one of the games in which the second virtual player defeated the first (which used the average weights from generation 0).

We also carried out 100 games between the best virtual player in generation 0 versus the best virtual player in generation 50, with a score of 85 to 15 in favor of the second virtual player. In Appendix B, we show one of the games in which the second virtual player defeated the first virtual player.

Additionally, we carried out 10 games between a virtual player which adopted the average weights from generation 50 and a (human) player ranked at 1600 points. The result was 9 to 1 in favor of the human player. Based on these played games, we used the Bayeselo tool[1] to estimate the ratings for both the human player and the chess engine using a minorization-maximization algorithm [15]. The obtained ratings are shown in Table III. In this table we can see that the rating obtained for the human player was 1737 and for the chess engine was 1463. In Appendix C, we can see the game that was won by the virtual player.

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

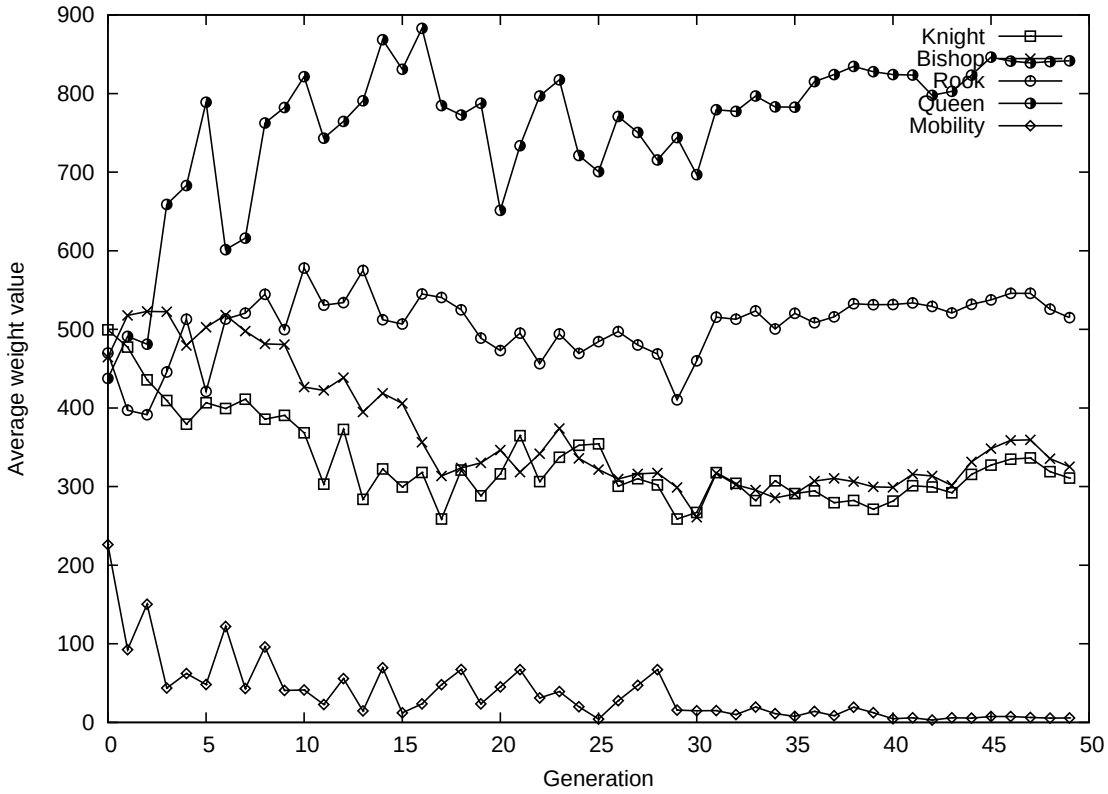

Fig. 2. Average weight values of the population during 50 generations.

It is worth indicating that in these games both virtual players used a database for openings and the depth of the search was set to 4 ply.

TABLE III
RATINGS FOR THE HUMAN PLAYER AND THE CHESS ENGINE IN A SIX GAMES MATCH. THE FINAL RESULT WAS 9 TO 1 FOR THE HUMAN PLAYER.

Rank	Name	Elo	+	-	Games	Score (%)	Oppo.	Draws (%)
1	Human player	1737	132	92	10	90%	1463	0%
2	Chess engine	1463	92	132	10	10%	1737	0%

VI. Conclusions and Future Work

We have reported here an evolutionary algorithm which incorporates a selection mechanism that favors virtual players that are able to "visualize" (or match) more movements from those registered in a database of chessmaster games. This information is used to tune the weights of our evaluation function, which is relatively simple to implement.

Our results indicate that the weight values obtained by our proposed approach closely match the known values from chess theory. Additionally, the standard deviations obtained from our runs were lower than those reported by other authors. Although similar weight values had been reported by other researchers, all of them had adopted tournaments between several players, contrasting with our approach, which is based on a database of grandmaster games.

As part of our future work, and aiming to create a chess program that will be able to play at the level of master or a ches master level, we plan to tune more weights (e.g., king security, doubled pawns, isolated pawns, past pawns, rooks in open columns, rooks in seventh row, control center of the board, and so on) using our proposed evolutionary algorithm. We plan to carry out more experiments varying the population size and increasing the number of games in the database. Additionally, we plan to use better strategies to explore the search space. Our aim is to increase the rating of our chess engine as much as we can, adopting relatively inexpensive approaches (computationally speaking).

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

In this Appendix, we show a game between the "average weights in generation 0" (with white pieces) versus the "average weights in generation 50" (with black pieces). The second virtual player won the game.

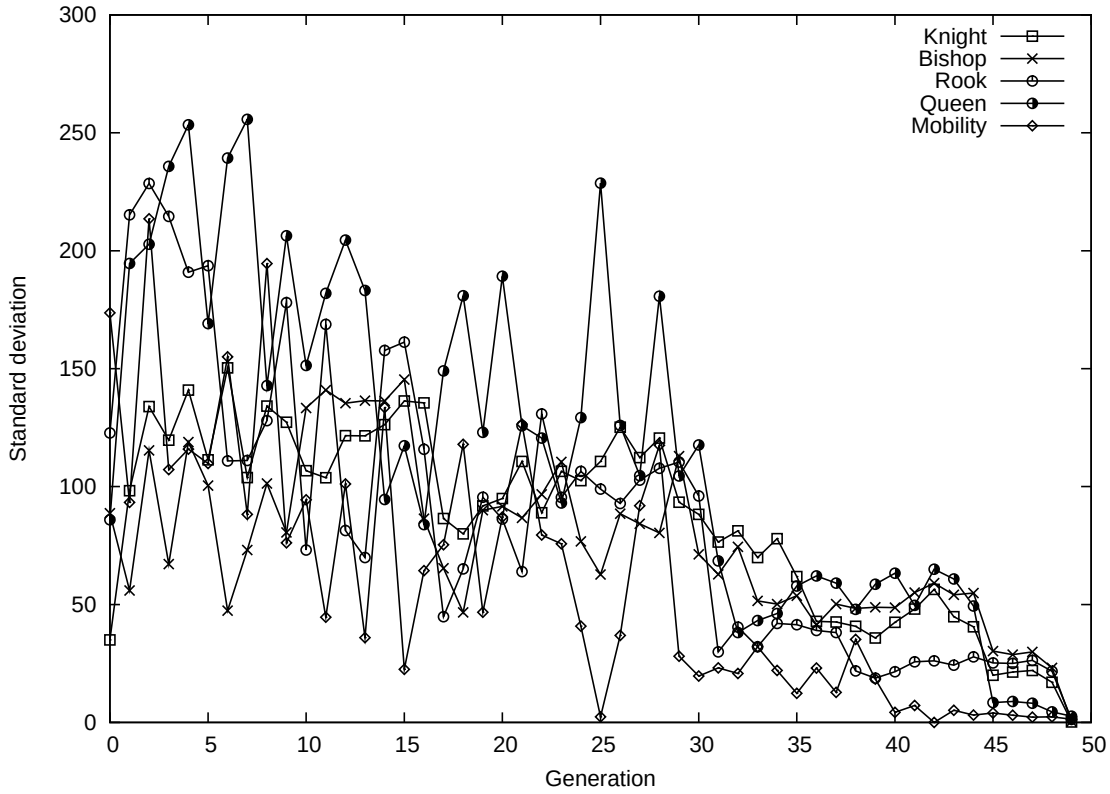

Fig. 3. Standard deviation of weights in the population during 50 generations.

[**White: "Average weights in generation 0"**]
[**Black: "Average weights in generation 50"**]
[**Result: "0-1"**]

1	**d4**	**d5**
2	**c4**	**c5**
3	N**c3**	N**f6**
4	**d**X**c5**	**d4**
5	N**b1**	N**c6**
6	**e3**	**e5**
7	**e**X**d4**	**e**X**d4**
8	N**f3**	BX**c5**
9	B**d3**	**O-O**
10	B**g5**	R**e8**+
11	B**e2**	Q**e7**
12	BX**f6**	**g**X**f6**
13	**a3**	**a5**
14	**a4**	B**f5**
15	N**a3**	**d3**
16	N**h4**	QX**e2**+
17	QX**e2**	RX**e2**+
18	K**d1**	B**g6**
19	NX**g6**	**f**X**g6**
20	R**f1**	R**ae8**
21	N**b5**	BX**f2**
22	N**c3**	R**e1**+
23	RX**e1**	RX**e1**+
24	K**d2**	RX**a1**
25	KX**d3**	N**e5**+
26	K**e2**	B**d4**
27	N**d5**	NX**c4**
28	NX**f6**+	BX**f6**
29	K**d3**	NX**b2**+
30	K**c2**	RX**a4**
31	**g3**	R**c4**+
32	K**b3**	K**f7**
33	**h4**	**b5**
34	**h5**	**g**X**h5**
35	**g4**	**h**X**g4**
36	K**a3**	**g3**
37	K**b3**	**g2**
38	K**a3**	**g1**Q
39	K**a2**	N**d3**
40	K**b3**	Q**b1**+
41	K**a3**	Q**b2**

++

APPENDIX B

In this Appendix, we show a game between "the best virtual player in generation 0" (with white pieces) versus "the best virtual player in generation 50" (with black pieces). The second virtual player won the game.

[**White: "The best virtual player in generation 0"**]
[**Black: "The best virtual player in generation** 50**"**]
[**Result: "0-1"**]

1	N**f3**	**d5**
2	**d4**	N**f6**
3	**c4**	**c6**
4	N**c3**	**e6**
5	**c5**	N**e4**
6	N**b1**	B**e7**
7	**b4**	Q**c7**
8	**g3**	**g5**
9	Q**d3**	N**d7**
10	B**h3**	**h5**
11	QX**e4**	**d**X**e4**
12	N**g1**	**a5**
13	B**g2**	**f5**
14	**b**X**a5**	QX**a5**+
15	B**d2**	Q**a4**
16	B**c3**	B**d8**
17	B**b2**	Q**c2**
18	B**a3**	B**a5**+
19	N**c3**	QX**c3**+
20	K**f1**	QX**a1**+
21	B**c1**	QX**c1**

++

APPENDIX C

In this Appendix, we show a game between a human player ranked at 1600 points (with white pieces) versus the "average weights in generation 50" (with black pieces). The virtual player won the game.

[**White: "Human player"**]
[**Black: "Average weights in generation 50"**]
[**Result: "0-1"**]

1	**c4**	**e5**
2	N**c3**	N**f6**
3	**e4**	B**b4**
4	N**ge2**	**O-O**
5	**h3**	**c6**
6	**a3**	BX**c3**
7	NX**c3**	**d5**
8	**c**X**d5**	**c**X**d5**
9	**e**X**d5**	**e4**
10	**g3**	B**f5**
11	B**g2**	Q**c8**
12	**f3**	**e**X**f3**
13	QX**f3**	R**e8**+
14	N**e2**	B**e4**
15	Q**f2**	BX**g2**
16	QX**g2**	N**bd7**
17	**O-O**	N**c5**

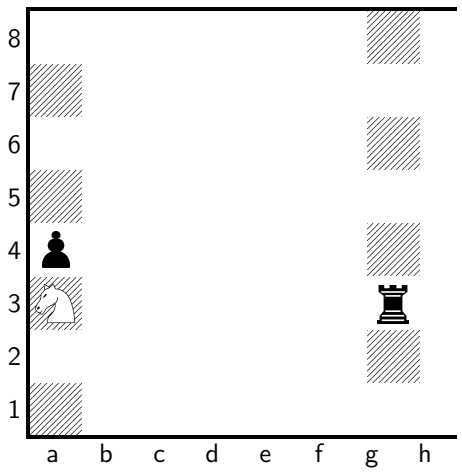

Fig. 4. Final position for the Appendix C game between human player ranked at 1600 points (with white pieces) versus "average weights in generation 50" (with black pieces).

18	**d4**	N**cd7**
19	B**g5**	**h6**
20	BX**f6**	NX**f6**
21	R**ac1**	Q**d7**
22	N**f4**	**b6**
23	R**f2**	R**ad8**
24	R**cf1**	R**e7**
25	**b4**	NX**d5**
26	N**h5**	N**e3**
27	Q**f3**	NX**f1**
28	RX**f1**	QX**d4**+
29	K**h2**	Q**e3**
30	**h4**	R**d2**+
31	K**h3**	QX**f3**
32	RX**f3**	**g6**
33	N**f6**+	K**g7**
34	N**g4**	**h5**
35	N**h2**	R**ee2**
36	N**f1**	R**a2**
37	**g4**	R**f2**
38	RX**f2**	RX**f2**
39	N**h2**	**h**X**g4**+
40	NX**g4**	R**f3**+
41	K**g2**	RX**a3**
42	N**e5**	R**a4**
43	N**c6**	K**h6**
44	K**g3**	K**h5**
45	**b5**	**f5**
46	N**e5**	**a5**
47	N**d7**	R**b4**
48	NX**b6**	RX**b5**
49	N**c4**	**a4**
50	N**a3**	R**b3**+

White resigns (see the final position in Figure 4).

REFERENCES

[1] D. F. Beal. A generalised quiescence search algorithm. *Artificial Intelligence*, 43(1):85–98, April 1990.

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

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

[4] 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.

[5] 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.

[6] 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.

[7] M. Campbell, A. J. Hoane, Jr., and F.-h. Hsu. Deep blue. *Artif. Intell.*, 134:57–83, January 2002.

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

[9] 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.

[10] 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.

[11] 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.

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

[13] D. Gomboc, M. Buro, and T. Marsland. Tuning evaluation functions by maximizing concordance. *Theoretical Computer Science*, 349(2):202–229, 2005.

[14] F. Hsu, T. Anantharaman, M. Campbell, and A. Nowatzyk. Deep thought. In *Computers, chess and cognition*, chapter 5, pages 55–78. Springer, Berlin, 1990.

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

[16] G. Kendall and 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*, volume 2, pages 995–1002. IEEE Press, May 2001.

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

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

[19] 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.

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

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

[22] S. Thrun. Learning to play the game of chess. In G. Tesauro, D. Touretzky, and T. Leen, editors, *Advances in Neural Information Processing Systems (NIPS) 7*, pages 1069–1076, Cambridge, MA, 1995. MIT Press.

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