An Adaptive Differential Evolution Algorithm with Opposition-Based Mechanisms, Applied to the Tuning of a Chess Program

Borko Bošković, Sašo Greiner, Janez Brest, Aleš Zamuda, and Viljem Žumer

University of Maribor
Faculty of Electrical Engineering and Computer Science
Smetanova 17, 2000 Maribor, Slovenia
borko.boskovic@uni-mb.si

Summary. This chapter describes an algorithm for the tuning of a chess program which is based on Differential Evolution using adaptation and opposition based optimization mechanisms. The mutation control parameter F is adapted according to the deviation of search parameters in each generation. Opposition-based optimization is included in the initialization, and in the evolutionary process itself. In order to demonstrate the behaviour of our algorithm we tuned our BBChess chess program with a combination of adaptive and opposition-based optimization. Tuning results show that adaptive optimization with an opposition-based mechanism increases the robustness of the algorithm and has a comparable convergence to the algorithm which uses only adaptation optimization.

Keywords: Differential Evolution, Adaptation, Tuning of a Chess Program, Opposition-Based mechanisms.

1 Introduction

Computer chess games have a long history of research in the field of artificial intelligence. Computer chess has advanced to a remarkable degree where computers now play against other computers and humans. With ever growing computer strength, we are witnessing more and more matches between computers and humans where computers usually win.

The reasons why the computer is beating humans are mainly hardware improvements and chess algorithm optimizations. The first computer that won against a human world champion chess player was Deep Blue which defeated the world champion chess player Garry Kasparov in 1996. In 2006 the Deep Fritz 10 computer program which ran on a PC, defeated world champion Vladimir Kramnik. So why are chess program developers trying to improve already very strong chess programs, even further? Many professional human chess players use chess programs to improve their own playing skills. Chess programs are also very useful in correspondence and freestyle chess. Matches between programs are also gaining popularity. As far as artificial intelligence is concerned, chess is regarded as a very useful environment for testing different approaches.

In this chapter we describe a Differential Evolution (DE) based algorithm for tuning chess programs. Using evolutionary concepts, this algorithm tunes chess programs and makes them stronger without any interaction with humans and without humans' expert knowledge. In order to improve the tuning process our algorithm includes the adaptation of DE control parameters and opposition-based optimization mechanisms. Because our DE uses adaptation and opposition-based optimization it is called 'AODE'.

The chapter is structured as follows. Section 2 gives an overview on tuning chess programs and briefly describes the basic DE and ODE (Opposition-Based Differential Evolution Algorithm). Section 3 describes the structure of those chess programs and parameters that may be tuned. Section 4 describes the details of our evolutionary algorithm AODE. Section 5 presents three experiments which tune the chess program by the use of AODE optimizations. We then show how these optimizations influence the tuning process. Section 6 concludes the chapter with final remarks.

2 Related Work

One of the possible improvements of a chess program is achieved by parameter tuning, but with conventional approaches this becomes a very difficult task. Developers have to change program parameters and then choose the best values through out the testing phase. The nature of such a task is very time consuming.

Another method is automated tuning or "learning". When we talk about automated tuning in computer chess we focus on algorithms such as hill climbing, simulated annealing, temporal difference learning [1, 2], and evolutionary algorithms [7, 8]. All approaches enable tuning on the basis of the program's own experiences, i.e. final result of a chess games competition: win, lose, or draw.

The pioneer of computer chess was Shannon (1949). He advocated the idea that computer chess programs would require an evaluation function and search algorithm to successfully play a game against human players [14]. In the beginning computer chess programs were designed "by hand" by the developers. The most important part of every chess program is its evaluation function. Evaluation functions contains a lot of parameters in the form of expressions and weights. In order to obtain a good evaluation function the developers had first to test it by playing numerous games and then modify it according to the produced results. Finding a proper evaluation function was a difficult and very time consuming task, because this was a recurring cycle. This is the main reason why current research has become involved in finding a method for automatically improving the evaluation function's parameters. Additionally, developers can tune the parameters of the search algorithm alone or together with those of the evaluation function.

Samuel [13] shows that a computer can be programmed so that it will learn to play better game of checkers than can be played by the person who wrote the program. The NeuroChess [17] is a program which learns to play chess from the final outcome of games. It learns its evaluation function, represented by artificial neural networks. This learning approach included inductive neural network learning, temporal differencing, and a variation of explanation-based learning. Another important work on learning is KnightCap [1] chess program. It learns parameters of its evaluation function using combination of Temporal Differences learning and on-line play on FICS and ICC chess servers. The program started with blitz rating 1650 and after 3 days of learning and 308 games played the program obtained blitz rating of 2150. The principles of evolution have also been used in the tuning of a chess evaluation function. Kendall and Whitwell [8] presented one such approach by using population dynamics. Fogel et al. [7] presented an evolutionary algorithm which has managed to improve a chess program by almost 400 rating points. Last two approaches used a population of individuals which consist of evaluation function parameters and new individuals are generated using mutation, crossover, and selection operators.

The DE [15, 9, 16] algorithm was proposed by Storn and Price, and since then it has been used in many practical cases. The original DE was modified and many new versions have been proposed [5]. Rahnamayan, Tizhoosh and Salama proposed an opposition-based DE (ODE) algorithm [11, 12]. ODE includes opposition based optimizations in order to improve the efficiency of classical DE algorithm. DE has also been used for chess program tuning [4, 6] because it converges quickly and improves playing ability during the evolutionary process.

3 Chess Program

The basic components of all modern chess programs are the search algorithm, evaluation function, move generator, transposition table, representation of game, opening book, and the end-game database [3]. These components enable a chess program to play equally well against the strongest human players, or even better. To improve an existing chess program with automated tuning, we can tune its parameters. The most tunable components are the evaluation function and the search algorithm.

The evaluation function contains a lot of expressions and parameters as weights of expressions. Expressions and parameters together represent all the chess knowledge of a chess program. Like the evaluation function, the search algorithm also contains parameters. However the number of parameters of a search algorithm is lower than in an evaluation function. The parameters are responsible for pruning the search tree and for selective searching.

Search algorithms only have a few parameters and their values have been tuned by the conventional approach (by hand and expert knowledge). On the other hand, an evaluation function has many more parameters which depend on each other and have been set by the developer according to experience and expert human instructions. Because an evaluation function contains complex expressions, the values of the parameters are approximated. Therefore, using automated tuning we can obtain better parameter values and improve the evaluation function and, consequently, the efficiency of chess programs.

4 AODE Algorithm for Tuning a Chess Program

Our tuning algorithm is based on Differential Evolution which uses adaptation and opposition-based optimization techniques. DE is a floating-point encoding evolutionary algorithm for global optimization over continuous spaces [9, 10]. Each generation of our AODE contains a current population P_g (g is a number of current generation), which further contains NP D-dimensional vectors (individuals) $\overrightarrow{X}_{g,i}$ with parameter values that represent the weights of a chess program.

$$\overrightarrow{X}_{g,i} = \{X_{g,i,1}, X_{g,i,2}, ..., X_{g,i,D}\},$$

$$i = 1, 2, ..., NP\ , \quad g = 1, 2, 3, ...$$

DE employs mutation, crossover, and selection operations during the evolutionary process, in each generation. Our algorithm uses the idea of adaptation and opposition-based optimization as shown in the algorithm below. P_0 represents the initial population, $P_{U,0}$ is an opposition population of P_0, P_1 is the first population, P_g is the current population, $P_{V,g}$ is the mutant population, $P_{U,g}$ is the trial population, and P_{g+1} is the population of the next generation. CR and JR are control parameters defined by the user.

Algorithm 1. AODE Algorithm

```
 1: Initialization($P_0$);
 2: $P_{U,0}$ = Opposition($P_0$);
 3: Evaluation($P_0, P_{U,0}, depth$);
 4: $P_1$ = Selection($P_0, P_{U,0}$);
 5: while continue tuning do
 6:   if rand(0,1) $<$ JR then
 7:     $P_{U,g}$ = DynamicOpposition($P_g$);
 8:   else
 9:     $P_{V,g}$ = AdaptiveMutation($P_g$);
10:     $P_{U,g}$ = Crossover($P_g, P_{V,g}, CR$);
11:   end if
12:   Evaluation($P_g, P_{U,g}$);
13:   $P_{g+1}$ = Selection($P_g, P_{U,g}$);
14: end while
```

4.1 Initialization

At the beginning, the population P_0 is initialized with parameter values that are distributed uniform-randomly between parameter bounds ($X_{j,low}, X_{j,high}; j = 1, 2, ..., D$). The bound values are problem-specific. In chess programs the parameters are set to approximate values by the developers. Developers can also intuitively determine those intervals which effectively define the bounds for parameters tuning. Accurately-defined bounds enable the algorithm to search through much smaller space and, consequently, find better parameters more quickly. If the search space is too limited, our algorithm can not find solution because it is out of bounds.

4.2 Opposition

The efficiency of the tuning process depends on the distance between the solution and the individuals in the initial population. After initialization, an opposite population $P_{U,0}$

is generated from the initial population. This mechanism together with evaluation (Section 4.3) and selection (Section 4.4), increases the probability of first generation containing individuals closer to the solution and, thus, accelerates convergence [11, 12].

The opposition population contains opposite individuals of the initial population and is defined by the following equations:

$$\overrightarrow{U}_{0,i} = \{U_{0,i,1}, U_{0,i,2}, ..., U_{0,i,D}\}$$

$$U_{0,i,j} = X_{j,low} + X_{j,high} - X_{0,i,j}$$

$$i = 1, 2, ..., NP\ , \quad j = 1, 2, ..., D,$$

where $\overrightarrow{U}_{0,i}$ represent the opposition individuals of the corresponding initial individuals $\overrightarrow{X}_{0,i}$.

4.3 Evaluation

Using trial $P_{U,g}(P_{U,0})$ and current $P_g(P_0)$ populations we have to evaluate their individuals. To do this we calculate the relative efficiencies of individuals according to both populations. Relative efficiency is measured according to the collected points and number of played games, as shown with the following equation:

$$\text{efficiency} = \frac{\text{collected points}}{2 \cdot \text{number of played games}}.$$

We can use more strategies to play games. Firstly, each individual of a trial population can play a specific number of games (N) against randomly chosen individuals of the current population. Secondly each individual of a trial population can play two games (one as white and one as black) against a corresponding individual of the current population. Other strategies are also possible.

An individual plays each game with a specific search depth and gets 2 points for winning, 1 for a draw, and 0 for losing. An individual wins when opponent's King is mate. The game is a draw if the position is a known draw position or the same position is obtained three times in one game, or because of the 50-moves rule. Games are limited to 150 moves for both players. Therefore, if the game has 150 moves, the result is a draw. An individual loses if its opponent wins.

4.4 Selection

The selection operation selects according to the relative efficiency of those individuals among the i-th current population and their corresponding individuals in the trial population. Selection dictates which individuals will survive into the next generation. In our case we used the following selection rule for a maximization problem:

$$\overrightarrow{X}_{g+1,i} = \begin{cases} \overrightarrow{U}_{g,i}, & \text{efficiency}(\overrightarrow{U}_{g,i}) > \text{efficiency}(\overrightarrow{X}_{g,i}), \\ \overrightarrow{X}_{g,i}, & \text{otherwise}. \end{cases}$$

$$i = 1, 2, ..., NP,$$

where $\overrightarrow{U}_{g,i}$ is an i-th individual from the trial population, $\overrightarrow{X}_{g,i}$ is an i-th individual from the current population, and $\overrightarrow{X}_{g+1,i}$ is an i-th individual from the population of the next generation.

4.5 Dynamic Opposition

As proposed in [11, 12], we can also use opposition-based optimization during the evolutionary process. This optimization is applied using a jump rate JR, as shown in Algorithm 1, and dynamic interval bounds ($X^g_{j,low}$, $X^g_{j,high}$; $j = 1, 2, ..., D$), as shown by the following equation:

$$\overrightarrow{U}_{g,i} = \{U_{g,i,1}, U_{g,i,2}, ..., U_{g,i,D}\}$$

$$U_{g,i,j} = X^g_{j,low} + X^g_{j,high} - X_{g,i,j}$$

$$i = 1, 2, ..., NP\ , \quad j = 1, 2, ..., D,$$

where $\overrightarrow{U}_{g,i}$ represents an opposition individual of a corresponding current individual $\overrightarrow{X}_{g,i}$ and $X^g_{j,low}$, $X^g_{j,high}$ are bound values for each parameter in the current population.

4.6 Adaptive Mutation

Adaptive mutation generates a mutant population $P_{V,g}$ from the current population P_g, using mutant strategy and adaptive mutation scale factor F. For each vector from the current population, mutation (using one of the mutation strategies) creates a mutant vector $\overrightarrow{V}_{g,i}$, which is an individual of mutant population.

$$\overrightarrow{V}_{g,i} = \{V_{g,i,1}, V_{g,i,2}, ..., V_{g,i,D}\}, \quad i = 1, 2, ..., NP.$$

DE includes various mutation strategies for global optimization. In our algorithm we used the $rand/2$ mutation strategy, which is given by the equation:

$$\overrightarrow{V}_{g,i} = \overrightarrow{X}_{g,r1} + F_g \cdot (\overrightarrow{X}_{g,r2} - \overrightarrow{X}_{g,r3}) + F_g \cdot (\overrightarrow{X}_{g,r4} - \overrightarrow{X}_{g,r5})$$

The indexes $r1$, $r2$, $r3$, $r4$, $r5$ are random and mutually different integers generated within the range $[1, NP]$ and also different from index i. F_g is a mutation scale factor in the g-th generation within the range $[0, 2]$ but usually less than 1.0. Because F_g scales the distance between the new and old individuals, it is responsible for exploration and exploitation balance in the evolutionary process. Therefore, we used adaptive F_g defined as the ratio of the standard deviations between parameters of the initial and current populations, as shown in the following equations:

$$F_g = \frac{\sum_{i=1}^{D} \sigma_{g,i}}{\sum_{i=1}^{D} \sigma_{0,i}}$$

$$\sigma_{g,i} = \sqrt{\frac{\sum_{j=1}^{NP}(X_{g,i,j} - \overline{X}_{g,i})^2}{NP - 1}}.$$

where $\sigma_{g,i}$ is a standard deviation of the i-th parameter in the current population.

4.7 Crossover

After mutation, a ”binary” crossover forms a trial population $P_{U,g}$. According to the i-th population vector and its corresponding mutant vector, crossover creates trial vectors $\overrightarrow{U}_{g,i}$ using the following rule:

$$\overrightarrow{U}_{g,i} = \{U_{g,i,1}, U_{g,i,2}, ..., U_{g,i,D}\}$$

$$U_{g,i,j} = \begin{cases} V_{g,i,j}, & rand_j(0,1) \leq CR \; or \; j = j_{rand}, \\ X_{g,i,j}, & otherwise. \end{cases}$$
$$i = 1, 2, ..., NP \;, \quad j = 1, 2, ..., D.$$

CR is a crossover factor within the range [0,1) and determines the probability of creating parameters of the trial vector from the mutant vector. Index j_{rand} is a randomly chosen integer within the range $[1, NP]$ and is responsible for the trial vector containing at least one parameter from the mutant vector. After crossover, the parameters of trial vector may be out of bounds ($X_{j,low}$, $X_{j,high}$). In this case the parameters can be mapped inside an interval, set to bounds or used as they are – out of bounds.

5 Experiments

Our algorithm was tested for tuning a simplified chess evaluation function of the chess program, BBChess. The evaluation function contains only material (values of pieces) and mobility (number of available moves for pieces) information, as shown in the following equation:

$$chess\_evaluation = X_m(M_w - M_b) + \sum_{i=0}^{5} X_i(N_{i,w} - N_{i,b}).$$

In this equation X_i represents material weights for all piece types without king and X_m the mobility weight. M_w represents mobility for white and M_b for black pieces. $N_{i,w}$ is the number of specific white pieces (i.e. the number of white pawns) and $N_{i,b}$ for specific black pieces. The principal reason for using such a simple and straightforward evaluation function was to demonstrate how the weight parameters of the function can be tuned by applying our tuning algorithm. In addition the behavior and features of the AODE algorithm were also presented. To do this, three experiments were performed. In the first experiment only adaptation optimization was used, in the second opposition-based optimization was included, and the third included all optimizations including opposition-based optimization during the evolutionary process.

In all experiments pawn material weight was fixed to 100 and the search depth set to 5 ply (half move). Experiments were run 30 times and tuning performed throughout 50 generations. The size of the population NP was 20 because larger NP would substantially increase the number of required games in one generation. The control parameter CR was set to 0.9 and the parameter bounds for all parameters were set to $X_{j,low} = 0$ and $X_{j,high} = 1000$ for all experiments. If the parameters were out of

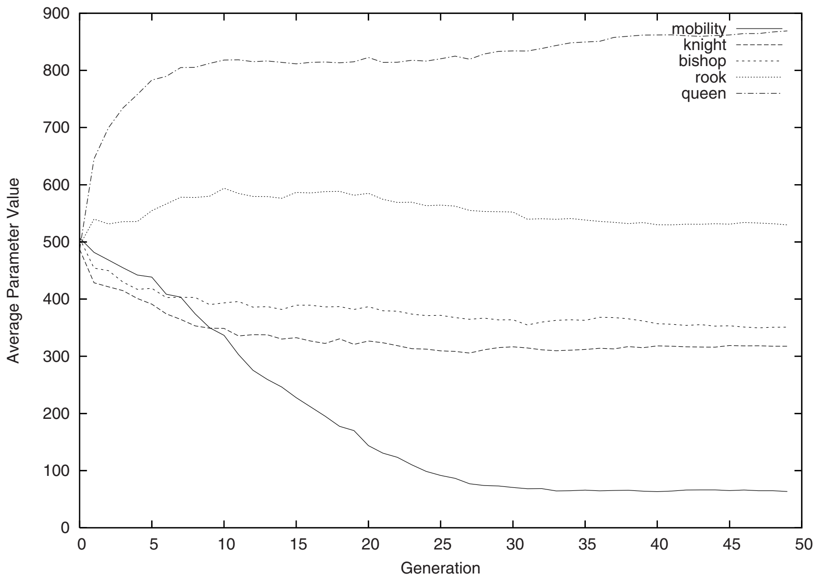

Fig. 1. Average parameter values along generations for AODE without opposition based optimization mechanisms

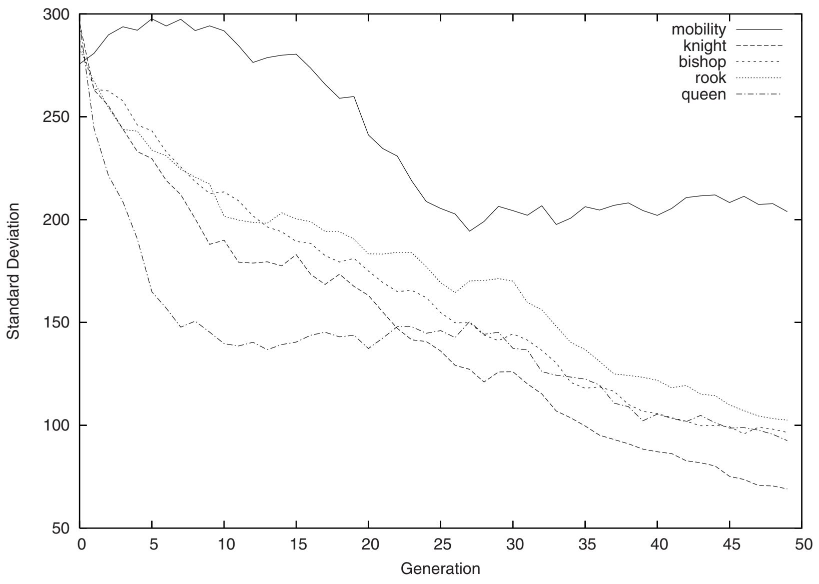

Fig. 2. Standard deviation of parameters along generations for AODE without opposition based optimization mechanisms

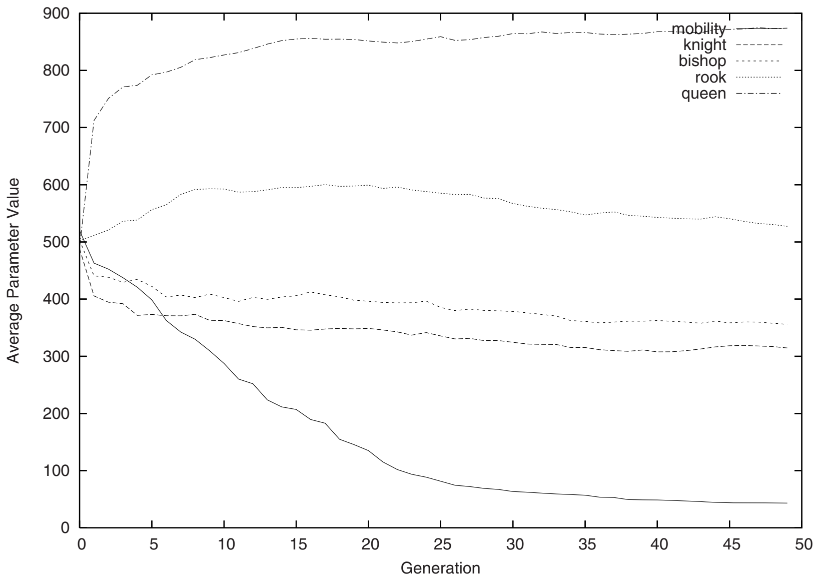

Fig. 3. Average parameter values along generations for AODE with initial population opposition and JR = 0.0

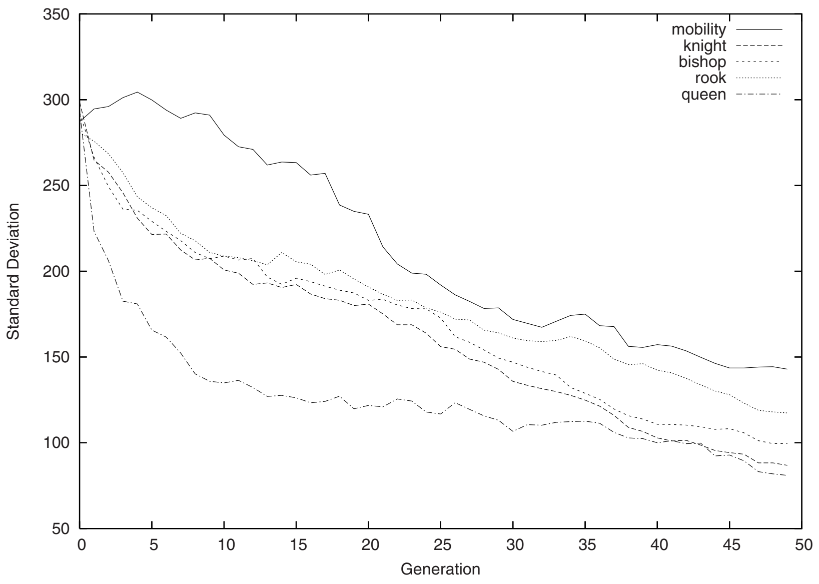

Fig. 4. Standard deviation of parameters along generations for AODE with initial population opposition and JR = 0.0

bounds after crossover they were set to bound values. When evaluating the individuals they evaluated according to two played games between corresponding individuals from the current and trial populations. Each individual played one game as a white player. This strategy was used because of the simplified evaluation function and it was that two games between corresponding individuals gave fair judgment as to which individual was better.

All experiments gave good parameter values. The number of runs was 30 for all experiments. Good parameter values are those which have an approximate ratio similar to that of the chess theory (Queen = 900, Rook = 500, Bishop = 330, Knight = 300 and mobility = 10).

The first experiment used only adaptive optimization (without opposition-based optimization during the evolutionary process $JR = 0.0$) and had average parameter values and standard deviation, as shown in Figures 1 and 2. Results of average parameter values show that the algorithm found good values. Standard deviation shows that our algorithm had some problems with tuning of mobility. The value of the mobility parameter greatly influenced the playing ability of a chess program. Large values mean that mobility becomes more important than material of pieces and generally speaking this weakens overall playing ability.

The second experiment included adaptive optimization and opposition-based optimization during initialization and had average parameter values and their standard deviation, as shown in Figures 3 and 4. This experiment also found good parameters values. The main difference from the first experiment is in the first few generations. In these generations the algorithm had better convergence.

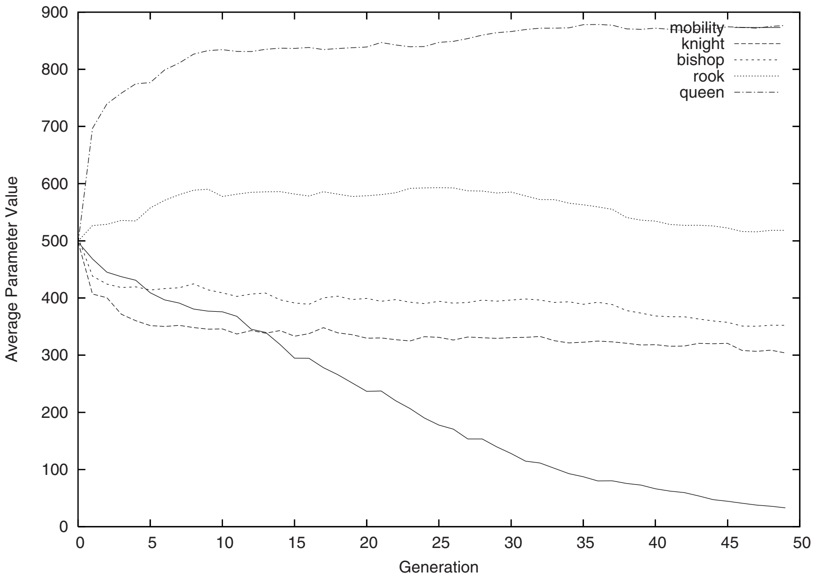

Fig. 5. Average parameter values along generations for AODE with initial population opposition and JR = 0.1

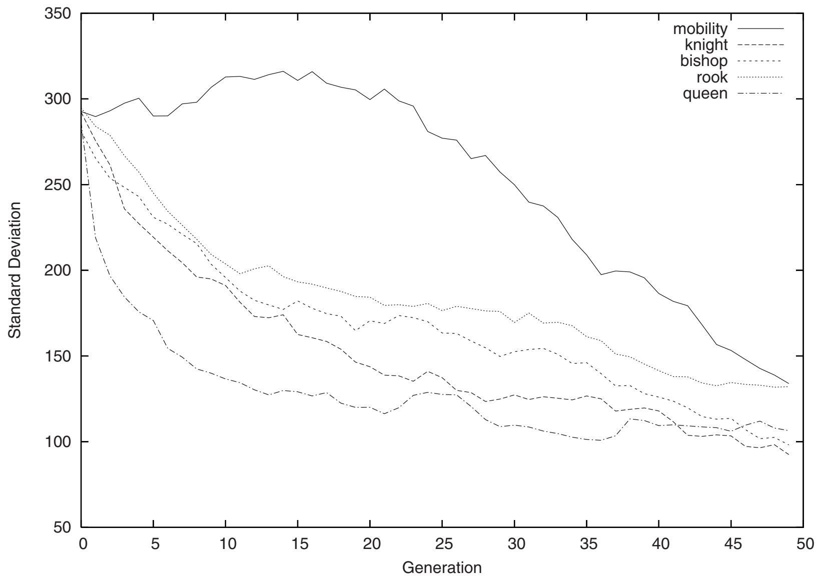

Fig. 6. Standard deviation of parameters along generations for AODE with initial population opposition and JR = 0.1

The third experiment included all optimizations ($JR = 0.1$) and achieved average parameter values and standard deviation are shown in Figures 5 and 6. In comparison with first two experiments this algorithm had poor convergence in the beginning but at the end obtained equally good parameters.

As shown on Figures 1, 3, and 5, the most critical parameter is mobility. Using additional analysis, it was discovered that the AODE in the first two experiments converged to local optima over two runs and in the third experiment only over one run. In all these runs, the algorithm found mobility parameter values that are considered inadequate in chess theory. In the third experiment we also observed that two populations had equal individuals sequentially because of repositioning in the dynamic opposition. Although JR was 0.1, the $rand(0, 1)$ was smaller than JR sequentially and, therefore, the algorithm generated a lot of equal individuals.

6 Conclusions

We have proposed an algorithm for the tuning of a chess program based on Differential Evolution. In the chess program we tuned only the parameters of its evaluation function. The algorithm included adaptation and opposition-based optimization mechanisms. Using different combinations of these mechanisms inside DE, which already includes adaptation, we have demonstrated the behavior of our algorithm. With opposition based-optimization only in initialization the algorithm has better convergence at the beginning. With opposition-based optimization during the entire evolutionary process the algorithm has poor convergence at the beginning but at the end obtains equal results. The results also show that such settings of the algorithm make it more robust.

References

1. Baxter, J., Tridgell, A., Weaver, L.: Experiments in Parameter Learning Using Temporal Differences. International Computer Chess Association Journal 21(2), 84–99 (1998)
2. Baxter, J., Tridgell, A., Weaver, L.: Learning to Play Chess Using Temporal Differences. Machine Learning 40(3), 243–263 (2000)
3. Bošković, B., Greiner, S., Brest, J., Žumer, V.: The Representation of Chess Game. In: Proceedings of the 27th International Conference on Information Technology Interfaces, pp. 381–386 (2005)
4. Bošković, B., Greiner, S., Brest, J., Žumer, V.: A Differential Evolution for the Tuning of a Chess Evaluation Function. In: CEC 2006: International Conference on Evolutionary Computation, Vancouver, Canada, July 2006, pp. 6742–6747. IEEE, Los Alamitos (2006)
5. Brest, J., Greiner, S., Bošković, B., Mernik, M., Žumer, V.: Self-Adapting Control Parameters in Differential Evolution: A Comparative Study on Numerical Benchmark Problems. IEEE Transactions on Evolutionary Computation 10(6), 646–657 (2006)
6. Chisholm, K.: Co-evolving Draughts Strategies with Differential Evolution, ch. 9, pp. 147–158. McGraw-Hill, London (1999)
7. Fogel, D.B., Hays, T.J., Hahn, S.L., Quon, J.: A Self-Learning Evolutionary Chess Program. Proceedings of the IEEE 92(12), 1947–1954 (2004)
8. Kendall, G., Whitwell, G.: An Evolutionary Approach for the Tuning of a Chess Evaluation Function Using Population Dynamics. In: Proceedings of the 2001 Congress on Evolutionary Computation CEC 2001, COEX, World Trade Center, 159 Samseong-dong, Gangnam-gu, Seoul, Korea, pp. 995–1002. IEEE Press, Los Alamitos (2001)
9. Price, K., Storn, R.: Differential Evolution: A Simple Evolution Strategy for Fast Optimization. Dr. Dobb's Journal of Software Tools 22(4), 18–24 (1997)
10. Price, K.V., Storn, R.M., Lampinen, J.A.: Differential Evolution, A Practical Approach to Global Optimization. Springer, Heidelberg (2005)
11. Rahnamayan, S., Tizhoosh, H.R., Salama, M.M.: Opposition-Based Differential Evolution Algorithms. In: CEC 2006: International Conference on Evolutionary Computation, Vancouver, Canada, July 2006, pp. 7363–7370. IEEE, Los Alamitos (2006)
12. Rahnamayan, S., Tizhoosh, H.R., Salama, M.M.A.: Opposition-based differential evolution. IEEE Transactions on Evolutionary Computation 12(1) (2008)
13. Samuel, A.L.: Some studies in machine learning using the game of checkers. IBM Journal of Research and Development (3), 211–229 (1995)
14. Shannon, C.: Programming a computer for playing chess. Philosophical Magazine 41(4), 256 (1950)
15. Storn, R., Price, K.: Differential Evolution - a simple and efficient adaptive scheme for global optimization over continuous spaces. Technical Report TR-95-012, Berkeley, CA (1995)
16. Storn, R., Price, K.: Differential Evolution – A Simple and Efficient Heuristic for Global Optimization over Continuous Spaces. Journal of Global Optimization 11, 341–359 (1997)
17. Thrun, S.: Learning To Play the Game of Chess. In: Tesauro, G., Touretzky, D., Leen, T. (eds.) Advances in Neural Information Processing Systems 7, pp. 1069–1076. The MIT Press, Cambridge (1995)