# An Adaptive Evolutionary Algorithm Based on Tactical and Positional Chess Problems to Adjust the Weights of a Chess Engine

Eduardo Vázquez-Fernández
ESIME-IPN
Carrera de Ingeniería en Computación
Av. Santa Ana No. 1000
Col. San Francisco Culhuacán
México D.F. 04430, MÉXICO
eduardovf@hotmail.com

Carlos A. Coello Coello
Computer Science Department
CINVESTAV-IPN
Evolutionary Computation Group
Av. IPN No. 2508, Col. San Pedro Zacatenco
México, D.F. 07360, MÉXICO
ccoello@cs.cinvestav.mx

***Abstract*—This paper employs an evolutionary algorithm to adjust the weights of the evaluation function of a chess engine. The selection mechanism of this algorithm chooses the virtual players (individuals in the population) that have the highest number of problems properly solved from a database of tactical and positional chess problems. This method has as its main advantage that we only mutate those weights involved in the solution of the current problem. Furthermore, the mutation mechanism is based on a Gaussian distribution whose standard deviation is adapted through the number of problems solved by each virtual player. We show here how, with the use of this method, we were able to increase the rating of our chess engine in $557$ Elo points (from $1760$ to $2317$).**

## I. Introduction

Computer chess is a topic that has attracted the interest of different scientists around the world. In $1947$, Alan Turing [28] designed a program to play chess. In $2012$, as part of the celebrations for his centenary birth, this program was finally implemented to play a game against the former world chess champion Garry Kasparov. In $1949$, at Bell Telephone Laboratories, Claude Shannon [26] proposed two strategies to implement a chess engine. "Strategy A" considered all possible moves to a fixed depth of the search tree (i.e., this is a brute force approach). "Strategy B" used chess knowledge to explore the main lines to a greater depth. Shannon was the first to estimate that the total number of possible chess games is $10^{120}$.

Computer chess has also attracted the interest of different companies around the world, including Bell Telephone Laboratories, Chessbase[1], and IBM, among others. The most significant event that related computer chess with companies was held in $1997$, when IBM's supercomputer *Deep Blue* defeated the reigning world champion in a classical chess match. This supercomputer was based mainly on Shannon's A strategy, and was capable of evaluating $200$ million positions per second. In spite of its importance, this event did not stop the research in this area. Today, chess programs focus more on Shannon's B strategy and usually include meta-heuristics in their search algorithms.

Chess is a game of perfect information between two adversaries. The number of different positions reachable from the beginning of the game is known as a *state-space*, and a *game tree* is a representation of the state-space of a chess game. In chess, the game tree consists of roughly $10^{43}$ nodes [26] so it is not possible to generate it completely. Instead, it is better to generate the *search tree* which is only a part of the game tree. A *principal variation* is a sequence of moves where both players play optimally. In order to find the principal variation in a chess game, it is necessary to evaluate the nodes without children (leave nodes) in the search tree through the evaluation function.

The main components of a chess engine are:

- The search algorithm.
- The move generator.
- The evaluation function.

Figure 1 shows the basic architecture of a chess engine. Next, we will briefly describe each of these components.

The *search algorithm* finds the principal variation from a given position on the board. The main algorithms that can be used in this component are: minimax, negamax, alpha-beta, negascout, quiescence, among others. The basic approach for games between two adversaries is the *minimax algorithm* [19], [22]. The *negamax algorithm* [6] is a more elegant implementation, which is also easier to program than the minimax algorithm because it applies the same operator at all levels in the tree. The main search method for games between two adversaries is the alpha-beta algorithm [18] which has the advantage of refraining from evaluating some nodes when unnecessary. It is also possible to use variants of the alpha-beta pruning algorithm such as the *negascout* [23] or the *Principal Variation Search* method [9]. The *quiescence algorithm* is used to extend the search tree to steady positions in which material exchanges, and king's checks (among others) cannot influence the resulting evaluation of a position.

Together with the search algorithm, the *evaluation function* is the most important part of a chess engine. As we saw above, it is only possible to represent the search tree down to a certain depth. Therefore, it is necessary to evaluate the leaf nodes through the evaluation function. This function is used to determine in a heuristic way the relative value of a position with respect to a particular side. The aim is that the evaluation function reflects the knowledge of the game.

The evaluation function is composed by a set of weights that store knowledge of the chess positions. A successful

[1] www.chessbase.com

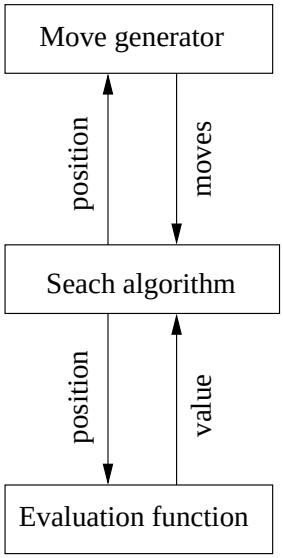


Fig. 1. Basic architecture of a chess engine.

adjustment of these weights allows a chess engine to play better. Developers of commercial chess programs must tune these weights using exhaustive manual test procedures. The main drawback of this method is the large amount of time (even years) needed to adjust the weights, and therefore the need to automate this task. In fact, this is the focus of the work reported in this paper. In Figure 1, the search algorithm invokes the evaluation function which returns a numerical value associated with this position.

The *move generator* generates all possible movements from a given position on the board. In Figure 1 the search algorithm invokes the move generator which returns the available moves on the current position.

The chess programs also use *hash tables* which store information about positions that had already been searched. Then, if the same position is reached again, no search is conducted, since the previously generated information would be used in that case.

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

## II. Previous Related Work

There are several papers in which the problems of adjusting the weights of a chess engine has been dealt with using co-evolution (tournaments among virtual players) (see [27], [12], [13], [4] [5], [3], [17], [21], and [29], among others).

This paper adopts chess problems to carry out the weights adjustment of a chess engine. Next, we will describe the works that make use of supervised learning to perform this sort of adjustment.

Gomboc et al. [15] applied an empirical gradient method to adjust the weights of a chess program. In this work, they used more than $600,00$ chess positions from a Chess Informant magazine to successfully adjust $11$ weights of the chess engine *Crafty* which is a state-of-the-art chess engine with a rating of $2614$ points.

David-Tabibi et al. [7] adjusted the weights of their chess engine with a genetic algorithm through reverse engineering (by mimicking the behavior of another chess program that served as a mentor). With this approach, they had an interesting participation in the $2008$ World Computer Chess Championship.

In [8], David-Tabibi et al. extended their previous related work [7]. Basically, they extended their previous experiments and carried out matches between the mentor and the evolved organisms. With their method, they obtained the sixth place in the $2008$ World Computer Chess Championship.

Vázquez-Fernández et al. [31] tuned the weights of their evaluation function through a database of chessmaster games. In this work, the chess material values obtained were similar to the values known from chess theory.

Vázquez-Fernández et al. [32] used exploration and exploitation to carry out the tuning of the weights of their chess engine. In the exploration step, they used an evolutionary algorithm with supervised learning. The selection mechanism of this algorithm uses games from chess grandmasters to decide which virtual player would pass to the next generation. This step is similar to their previous related work [31] with the difference that now they adjusted a larger number of weights (form $5$ to $29$ weights). With this method, they obtained an increase in the rating of their chess engine from $1463$ to $2205$. In the exploitation step, they used the Hooke-Jeeves algorithm to continue the adjustment of the weights for the best virtual player obtained in the previous step. Using this algorithm as a local search engine, they increased the rating of their chess engine from $2205$ to $2425$ points.

Vázquez-Fernández et al. [30] used tactical chess problems to adjust the weights of their chess engine. This method allows to mutate only those weights involved in the current problem, preventing mutations that can lead to incorrect values for future evaluations of board positions. Such a method adapts the mutation rate based on the number of problems that have been solved for each virtual player.

The present work is a continuation of our previous related work [30], in which the main differences are:

- Before, we only used tactical chess problems. Now, we add positional chess problems to our database.
- We consider now the weights associated with the bishop's and queen's positional value.
- Experimentally, we found the ideal number of virtual players within the range $[8, 50]$.
- We use a different mutation operator. In our previous work, we used Michalewicz's non-uniform mutation operator [20]. Now, we employ a mutation operator based on a Gaussian distribution because we gathered empirical evidence that indicates the superiority of

this operator with respect to Michalewicz's mutation operator.

It is important to mention that with these modifications, we were able to increase the rating of our chess engine from 1760 (see [30]) to 2317 (see Section V) rating points. The concept of rating in a chess engine is explained in Appendix A.

## III. Our Chess Engine

For a particular side, we evaluated a given position on the board with the following expression:

$$eval = materialValue + positionalValue \tag{1}$$

where:

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

$X_i$ represents the material value for piece $i$, and $r$ is the number of pieces of one side in particular, regardless of the king.

On the other hand,

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

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

The king 's positional value is given by:

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

where:
$X_{king,i}$ is the weight of factor $F_{king,i}$ (a *factor* is a positional characteristic of a particular piece, for example, its mobility).
$F_{king,1}$ is the sum of material values of pieces that defend their king.
$F_{king,2}$ is the sum of material values of pieces that attack the king.
$F_{king,3}$ is true if the king is castled; otherwise, it is false.
$F_{king,4}$ is the number of pawns that protect their king.

The queen's positional value is given by:

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

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

The rook's positional value is given by:

$$P_{Rook} = \sum_{i=1}^{4} X_{rook,i} * F_{rook,i} \tag{6}$$

where:
$X_{rook,i}$ is the weight of factor $F_{rook,i}$.
$F_{rook,1}$ is the mobility of the rook.
$F_{rook,2}$ is true if the rook is on an open column; otherwise, it is false.
$F_{rook,3}$ is true if the rook is on the seventh row; otherwise, it is false.
$F_{rook,4}$ is true if there are two rooks on the seventh row; otherwise, it is false.

The bishop's positional value is given by:

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

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

The knight's positional value is given by:

$$P_{Knight} = \sum_{i=1}^{4} X_{Knight,i} * F_{Knight,i} \tag{8}$$

where:
$X_{Knight,i}$ is the weight of factor $F_{Knight,i}$.
$F_{Knight,1}$ is the mobility of the knight.
$F_{Knight,2}$ is true if the knight is in the periphery of the board; otherwise, it is false.
$F_{Knight,3}$ is true if the knight is defended by a pawn; otherwise, it is false.
$F_{Knight,4}$ is true if the knight cannot be evicted by an enemy pawn; otherwise, it is false.

The pawn's positional value is given by:

$$P_{Pawn} = \sum_{i=1}^{4} X_{Pawn,i} * F_{Pawn,i} \tag{9}$$

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

The material value of a piece is a static value. Shannon [26] assigned 100, 300, 330, 500 and 900 points for the pawn, knight, bishop, rook and queen, respectively. In this work, the pawn's material value is always 100.

The positional value of a piece is a dynamic value and depends on many factors such as mobility, board location,

strength, etc. In other related works (for example [12]), the positional value of a piece only depends on its board location because these values are stored in arrays of $64$ squares. The idea of our proposal is that the chess positional values depend directly on the characteristics of the position. It is expected that while more features are taken into account in calculating the positional value of a piece, this value will be more accurate, and therefore, the position will be better evaluated.

The purpose of this paper is to tune the weights of equations (2), (4), (5), (6), (7), (8) and (9) using evolutionary programming [14] and a database of positional and tactical chess problems extracted from [24] [10], [25] and [1]. The aim is that the adjustment of the weights performed by our approach leads to an increase in the rating of our chess engine.

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

- Election of movements through the alpha-beta algorithm [18].
- Stabilization of positions through the Quiescence algorithm [2] that takes into account the exchange of material and king's checks.
- Use of hash tables [33].

In these experiments, our chess engine used the database *Olympiad.abk* in the opening phase. This database is included with the graphical user interface *Arena*.[2]

## IV. Our Proposed Approach

In our previous related work [30], we used tactical chess problems to adjust the weights of our chess engine. Now, we use tactical and positional chess problems to adjust these weights. Under this approach, the virtual players improve their playing strength with respect to our previous work.

The evolutionary algorithm adopted in [30] adjusts the weights of $N$ virtual players, so that the virtual players with more problems properly solved acquire the right to pass to the next generation. The idea is the following: each virtual player is asked if it can solve the current problem. If so, the player increases in one its number of problems solved. Once all the virtual players have been asked about the current solution, the $N/2$ virtual players which have properly solved the highest number of problems are selected, and they become eligible to mutate the remaining half. This process is carried out for $Gmax = 50$ generations or until all the virtual players have solved the current problem. The algorithm was tested for $N = 8, 9, \ldots, 50$ virtual players, and for each of these values, it was tested for 30 to 60 problems in the database (half of the problems are tactical in nature, and the other half are positional in nature).

Algorithm 1 shows the evolutionary algorithm used to adjust the weights of our chess engine. Line 1 gets the set $P$ which consists of $numP$ problems chosen at random from the database of problems $S$. Half of these problems are tactical in nature, and the other half are positional in nature. Line 2 chooses a particular problem $p$ from the set $P$. In lines 3 to 5, we establish that the solution of the problem $p$ and the number of problems that have been solved is zero for each virtual player. Line 7 sets the weights that can be mutated or not (because of the importance of this part of the algorithm, in Section IV we discuss it in more detail). Line 8 sets the starting position of the problem $p$. Line 9 returns the solution $m$ of the problem $p$. Line 10 sets the generation counter $g$ equal to zero. In lines 11 to 23 we carry out the adjustment of the weights for the problem $p$ during $Gmax$ generations. In lines 11 to 18, each virtual player computes its next move $n$, and if this movement matches the movement $m$, then this virtual player establishes that it has found the solution to the problem $p$ and it increases its number of problems solved in 1. In lines 19 to 21 we go to the next problem if all the virtual players have found the solution to the problem $p$. In line 22 we apply the selection mechanism and in line 23, we apply the mutation operator.

**Algorithm 1** EvolutionaryAlgorithm()

```
1:  P ← chooseProblems(S, numP)
2:  for each problem p in P do
3:    for i = 1 → N do
4:      foundSolution[i] ← FALSE
5:      solutions[i] ← 0
6:    end for
7:    establishWeightsToMutate()
8:    setPosition(p)
9:    m ← solution(p)
10:   g ← 0
11:   while g++ < Gmax do
12:     for i = 1 → N do
13:       n ← nextMovement(i)
14:       if m == n then
15:         foundSolution[i] = TRUE
16:         solutions[i]++
17:       end if
18:     end for
19:     if allProblemsFoundSolution()==TRUE then
20:       break
21:     end if
22:     selection()
23:     mutation()
24:   end while
25: end for
```

### *Initialization*

In our previous related work [30], the initial population consisted of $N = 8$ virtual players. Now, we use $N = 8, 10, \ldots, 20$ virtual players (we choose even values to have $N/2$ parents and $N/2$ offspring in subsequent generations). The weights of the virtual players were randomly initialized within their allowable bounds using a uniform distribution. These bounds were defined by a chess expert. The left and right bounds ($X_{j,low}$ and $X_{j,high}$, respectively) are shown in Table I.

### *Selection*

The selection mechanism of this step chooses the $N/2$ virtual players having the highest number of problems properly solved, and these virtual players are mutated to generate the remaining $N/2$ virtual players.

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

TABLE I. THIS TABLE SHOWS THE WEIGHTS, THE LEFT BOUND OF THE RANGE, THE RIGHT BOUND OF THE RANGE, THE AVERAGE WEIGHT VALUE AND THEIR STANDARD DEVIATION FOR THE WEIGHT $j$ AT THE END OF THE EVOLUTIONARY PROCESS FOR THE BEST RUN. THE BOUND VALUES WERE CHOSEN BY AN EXPERT IN CHESS.

| $X_j$ | $X_{j,low}$ | $X_{j,high}$ | Value | $\sigma$ |
|---|---|---|---|---|
| $X_{pawn}$ | 0 | 0 | 100.00 | 0.00 |
| $X_{knight}$ | 200 | 400 | 297.29 | 0.00 |
| $X_{bishop}$ | 200 | 400 | 312.16 | 0.00 |
| $X_{rook}$ | 400 | 600 | 493.57 | 19.44 |
| $X_{queen}$ | 800 | 1000 | 907.25 | 18.32 |
| $X_{king,F_1}$ | $-100$ | 100 | 78.83 | 9.27 |
| $X_{king,F_2}$ | $-100$ | 100 | $-84.82$ | 6.67 |
| $X_{king,F_3}$ | $-100$ | 100 | 61.230 | 6.56 |
| $X_{king,F_4}$ | $-100$ | 100 | 81.12 | 5.56 |
| $X_{queen,F_1}$ | $-100$ | 100 | 9.14 | 2.01 |
| $X_{rook,F_1}$ | $-100$ | 100 | 22.44 | 9.87 |
| $X_{rook,F_2}$ | $-100$ | 100 | 49.22 | 16.56 |
| $X_{rook,F_3}$ | $-100$ | 100 | 16.23 | 18.09 |
| $X_{rook,F_4}$ | $-100$ | 100 | 72.22 | 3.07 |
| $X_{bishop,F_1}$ | $-100$ | 100 | 12.03 | 2.45 |
| $X_{knight,F_1}$ | $-100$ | 100 | 74.25 | 2.34 |
| $X_{knight,F_2}$ | $-100$ | 100 | $-51.67$ | 3.25 |
| $X_{knight,F_3}$ | $-100$ | 100 | 19.34 | 4.07 |
| $X_{knight,F_4}$ | $-100$ | 100 | 84.56 | 2.29 |
| $X_{pawn,F_1}$ | $-200$ | 200 | $-133.92$ | 10.76 |
| $X_{pawn,F_2}$ | $-200$ | 200 | $-78.56$ | 18.32 |
| $X_{pawn,F_3}$ | $-200$ | 200 | 131.19 | 15.09 |
| $X_{pawn,F_4}$ | $-200$ | 200 | 43.33 | 12.65 |

*Mutation Operator*

One offspring was created from each surviving parent by mutating all weights in equations (2), (4), (5), (6), (7), (8) and (9).

In our previous related work [30], we adopted Michalewicz's non-uniform mutation operator [20]. In this paper, we employ a mutation based on a Gaussian distribution because in all the experiments that we performed, the Gaussian operator outperforms Michalewicz's mutation operator.

Mutation was implemented according to:

$$X_i^{'} = X_i + N(\mu, \sigma) \tag{10}$$

where $N(\mu, \sigma)$ is a Gaussian random variable with mean $\mu$ and standard deviation $\sigma$. The mean is given by the midpoint of the range of the weight $X_i$, that is:

$$\mu = (X_{j,low} + X_{j,high})/2 \tag{11}$$

where $X_{j,low}$ and $X_{j,high}$ are the left and right bounds of the weight $X_i$, respectively. The standard deviation is given by:

$$\sigma = 3(-solutions[i]/numP + 1)\mu \tag{12}$$

where $solutions[i]$ denotes the number of problems solved by the virtual player $i$, and $numP$ denotes the number of problems chosen from the database for adjusting the weights. So, the expression $(-solutions[i]/numP+1)$ is equal to one (or zero) if the virtual player $i$ has not resolved any problem from the database (or has resolved $numP$ problems from the database). With this contribution to the present work, the mutation of the weight $X_i$ is adapted through the evolutionary process. The idea of the term $3\mu$ is cover practically all the Gaussian bell at the beginning of the evolutionary process.

*Database of Games*

In our experiments, we used a database consisting of $400$ chess tactical problems and $400$ chess positional problems. The tactical problems were taken from [24], and the positional problems were taken from [10], [25] and [1].

## V. EXPERIMENTAL RESULTS

We carried out two experiments. In the following subsections we describe them in detail.

### A. Experiment A

In the first experiment, we tuned the weights of equations (2), (4), (5), (6), (7), (8) and (9). These weights were random values generated with a uniform distribution within their allowable bounds. If, after mutation, the weight $X_j$ falls, either to the left or to the right of the allowable range $[X_{j,low}, X_{j,high}]$, then its value is set to $X_{j,low}$, or to $X_{j,high}$, respectively. The number of virtual players $N$ took values in the range $[8, 50]$ (we choose even values to have $N/2$ parents and $N/2$ offspring in subsequent generations), and the number of training chess problems $numP$ took even values in the range from $[30, 100]$ (we choose even values to have $numP/2$ tactical chess problems and $numP/2$ positional chess problems).

This experiment consisted of performing twenty runs for each value of $N$ combined with each value of $numP$. At the end of each run, we carried out $200$ games between the evolved virtual player and the non-evolved virtual player. Table II shows the best $20$ results in which the evolved virtual player achieved the best percentage of victories against the non-evolved virtual player. The runs are sorted in descending order based on the percentage of victories (column $Wins\%$), so the best result is shown in row $1$. In this row, the evolved virtual player won $189$, drew $11$ and lost $0$ games against the non-evolved virtual player (the percentage of games won by the evolved virtual player was $97.25\%$). We can see in this table that the evolved virtual player always exceeded the winning percentage of the non-evolved virtual player by a wide margin. We can also see that the ideal number of virtual players was in the range from $18$ to $26$, where $22$ was the most frequent value (with eight repetitions). In this table, we can see that the highest values of the variable $numP$ are in the best $20$ runs. Therefore, it is expected that if we increase the number of training chess problems in this variable, the winning percentage of the evolved virtual player with regard to the non-evolved virtual player will also be increased.

In this experiment we used a search depth of four plies (1 ply corresponds to the movement of one side).

TABLE II. NUMBER OF GAMES WON, DRAWN AND LOST FOR THE BEST VIRTUAL PLAYER IN GENERATION 50 AGAINST THE BEST VIRTUAL PLAYER AT GENERATION 0.

| Row | Wins | Draws | Losses | Wins% | N | numP |
|---|---|---|---|---|---|---|
| 1 | 189 | 11 | 0 | 97.25% | 20 | 96 |
| 2 | 189 | 10 | 1 | 97.00% | 22 | 98 |
| 3 | 188 | 12 | 0 | 97.00% | 22 | 98 |
| 4 | 188 | 12 | 0 | 97.00% | 20 | 96 |
| 5 | 188 | 12 | 0 | 97.00% | 24 | 92 |
| 6 | 187 | 13 | 0 | 96.75% | 18 | 90 |
| 7 | 187 | 13 | 0 | 96.75% | 20 | 96 |
| 8 | 187 | 13 | 0 | 96.75% | 22 | 88 |
| 9 | 188 | 11 | 1 | 96.75% | 20 | 92 |
| 10 | 186 | 14 | 0 | 96.50% | 22 | 94 |
| 11 | 186 | 13 | 1 | 96.25% | 24 | 94 |
| 12 | 186 | 13 | 1 | 96.25% | 22 | 92 |
| 13 | 185 | 14 | 1 | 96.00% | 22 | 96 |
| 14 | 184 | 16 | 0 | 96.00% | 24 | 96 |
| 15 | 185 | 13 | 2 | 95.75% | 22 | 94 |
| 16 | 184 | 15 | 1 | 95.75% | 20 | 98 |
| 17 | 185 | 13 | 2 | 95.75% | 24 | 98 |
| 18 | 184 | 15 | 1 | 95.75% | 26 | 96 |
| 19 | 183 | 16 | 1 | 95.50% | 20 | 90 |
| 20 | 183 | 15 | 2 | 95.25% | 22 | 86 |

### B. Experiment B

In this experiment, the non-evolved virtual player was called $VP_{non-evolved}$ and played 200 games against the chess program *Rybka 2.3.2a* using each of the following ratings: 2500, 2300, 2100 and 1900. A histogram of our results is shown in Figure 2. For example, $VP_{non-evolved}$ won, drew and lost 0, 0 and 200, respectively, against *Rybka 2.3.2a* at 2500 rating points; $VP_{non-evolved}$ won, drew and lost 0, 2 and 198, respectively, against *Rybka 2.3.2a* at 2300 rating points. The same experiment was carried out with the evolved virtual player which corresponds to the first row in Table II. This virtual player was called $VP_{evolved}$. The histogram of the results is shown in Figure 3. In this Figure, we can see that $VP_{evolved}$ won, drew and lost 29, 40 and 131, respectively, against *Rybka 2.3.2a* at 2500 rating points; $VP_{evolved}$ won, drew and lost 77, 48 and 75, respectively, against Rybka 2.3.2a at 2300 rating points, and so on.

Based on these played games, we used the Bayeselo tool[3] to estimate the ratings of the virtual players and *Rybka 2.3.2a* using a minorization-maximization algorithm [16]. The obtained ratings are shown in Table III. In this table we can see that the rating for the virtual player $VP_{non-evolved}$ was 1501, and the rating for virtual player $VP_{evolved}$ was 2317, representing an increase of 816 rating points between the non-evolved and the evolved virtual players after the evolutionary process for the first run in Table II.

It is worth mentioning that these experiments were also carried out by using Michalewicz non-uniform mutation operator instead of the mutation operator based on a Gaussian distribution. In this case, the best evolved virtual player registered a rating of 2282 points against *Rybka 2.3.2a*.

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

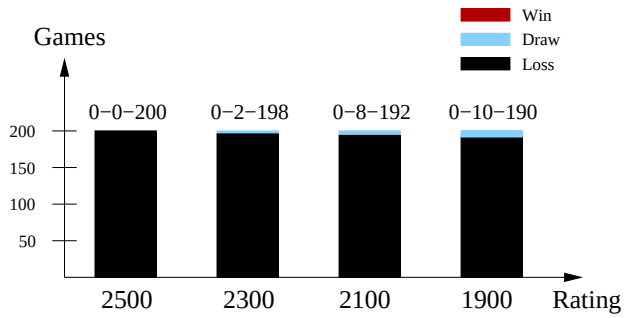


Fig. 2. Histogram of wins, draws and losses for the non-evolved virtual player against *Rybka 2.3.2a*.

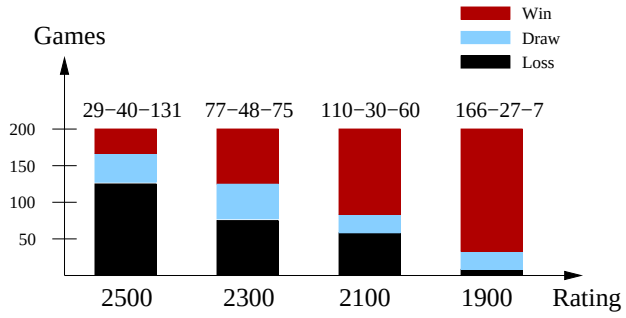


Fig. 3. Histogram of wins, draws and losses for the evolved virtual player against *Rybka 2.3.2a*.

## VI. CONCLUSIONS AND FUTURE WORK

In this paper, we presented an evolutionary algorithm to adjust the weights of the evaluation function of a chess engine. The selection mechanism of this algorithm gives priority to the virtual players who had properly solved more chess problems from our database.

In our previous related work, we only used tactical chess problems. Now, we added positional chess problems to our database. Also, we added the weights associated with the bishop's and queen's positional value. The mutation mechanism was also modified. Before, this mechanism was based on Michalewicz's non-uniform mutation operator. Here, the mutation mechanism used a Gaussian distribution whose standard deviation is adapted through the number of problems solved by each virtual player. With these changes, we increased the rating of our chess engine in 557 rating points (from 1760 to 2317).

From our experiments, we concluded that the ideal number of virtual players was in the range from 18 to 26, where 22 was the most frequent value. We also found that, as one would

TABLE III. RATINGS OF THE VIRTUAL PLAYERS AND *Rybka2.3.2a*.

| Rank | Name | Elo | + | - | Games | Score (%) | Oppo. | Draws (%) |
|---|---|---|---|---|---|---|---|---|
| 1 | $Rybka_{2500}$ | 2510 | 39 | 37 | 400 | 88% | 1909 | 10% |
| 2 | $VP_{evolved}$ | 2317 | 19 | 19 | 800 | 57% | 2256 | 18% |
| 3 | $Rybka_{2300}$ | 2312 | 34 | 34 | 400 | 75% | 1909 | 13% |
| 4 | $Rybka_{2100}$ | 2207 | 35 | 36 | 400 | 68% | 1909 | 10% |
| 6 | $Rybka_{1900}$ | 1994 | 38 | 39 | 400 | 54% | 1909 | 9% |
| 5 | $VP_{non-evol}$ | 1501 | 52 | 62 | 800 | 1% | 2256 | 3% |

expect, as more chess problems are used in the training phase, the strength of the virtual players gets better.

It is worth mentioning that the material values of the chess pieces are similar to the values known from chess theory.

As part of our future work, we plan to add weights to the evaluation function of our chess engine in order to increase its rating as much as we can. Also, we plan to use better strategies that allow us a more efficient exploitation and exploration of the search space. Similarly, it would be interesting to test our method with other mutation operators as well as with other evolutionary algorithms such as differential evolution, evolution strategies, etc.

## ACKNOWLEDGEMENTS

The first author acknowledges the National Polytechnical Institute (IPN). The second author acknowledges support from CONACyT project no. 103570.

## APPENDIX A

The Elo rating system is a method for calculating the relative strength of players in games with two opponents such as chess. In this system, each player has a numerical rating, and a higher number denotes a higher playing strength of the player concerned.

The formula to obtain the Elo rating of a player is given by [11]:

$$R_{new} = R_{old} + K(outcome - W), \tag{13}$$

where:
$R_{new}$ is the new rating.
$R_{old}$ is the old rating.
$K$ is a constant that depends on the rating.
$outcome$ is the game result.
$W$ is the expected or percentage score given by the logistic curve.

The $outcome$ is given by:

$$outcome = \begin{cases} 1, & \text{for a win} \\ 0.5, & \text{for a draw} \\ 0, & \text{for a loss} \end{cases}$$

The expected or percentage score $W$ is given by:

$$W = \frac{1}{1 + 10^{\frac{R_{opponent} - R_{old}}{400}}}, \tag{14}$$

here $R_{opponent}$ is the opponent's rating.

This method was created by the mathematician Arpad Elo, and has been adopted by the United States Chess Federation (USCF) since $1960$ and by the Fédération Internationale des Échecs (FIDE) since $1970$. Table IV shows the classification of the USCF.

TABLE IV. 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 |

## REFERENCES

[1] J. Aagaard. *Excelling at Positional Chess*. Everyman Chess, 1st edition edition, August 2003.

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

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

[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] M. S. Campbell and T. A. Marsland. A comparison of minimax tree search algorithms. *Artificial Intelligence*, 20(4):347–367, 1983.

[7] O. David-Tabibi, M. Koppel, and N. S. Netanyahu. Genetic algorithms for mentor-assisted evaluation function optimization. In *Proceedings of the 10th annual conference on Genetic and evolutionary computation*, GECCO '08, pages 1469–1476, New York, NY, USA, 2008. ACM.

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

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

[10] A. Dunnington. *Can You Be a Positional Chess Genius*. Everyman Chess, 1st edition edition, August 2002.

[11] A. E. Elo. *The rating of chessplayers, past and present*. Arco Pub., New York, 1978.

[12] 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.

[13] 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.

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

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

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

[17] 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.

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

[19] T. A. Marsland. A review of game-tree pruning. *International Computer Chess Association Journal*, 9(1):3–19, 1986.

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

[21] 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.

[22] A. Newell, J. C. Shaw, and H. A. Simon. Chess-playing programs and the problem of complexity. *IBM J. Res. Dev.*, 2:320–335, October 1958.

[23] A. Reinefeld. An improvement of the scout tree-search algorithm. *Chess Association Journal*, 4(6):4–14, 1983.

[24] F. Reinfeld. *One Thousand and One Winning Chess Sacrifices and Combinations*. Wilshire Book Company, 1969.

[25] P. P. Robert Bellin. *Test Your Positional Play*. Collier Books, first edition edition, December 1985.

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

[27] 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.

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

[29] E. Vázquez-Fernández, C. Coello, and F. Sagols. Assessing the positional values of chess pieces by tuning neural networks' weights with an evolutionary algorithm. In *Proceedings of 2012 World Automation Congress (WAC 2012)*, Puerto Vallarta, México, 2012.

[30] E. Vázquez-Fernández, C. A. C. Coello, and 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, pages 39–40, Dublin, Ireland, July 12–16 2011. ACM.

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

[32] E. Vázquez-Fernández, C. A. C. Coello, and F. D. S. Troncoso. An evolutionary algorithm coupled with the hooke-jeeves algorithm for tuning a chess evaluation function. In *IEEE Congress on Evolutionary Computation*, pages 1–8, Brisbane, Australia, 2012.

[33] A. Zobrist. A new hashing method with application for game playing. Technical Report 88, The University of Wisconsin, Madison WI, USA, 1970. Reprinted (1990) in ICCA Journal, Vol. 13, No. 2, pp. 69-73.