Complex Systems Seminar
Can complex systems help design better chess-playing programs?

Olivier Mehani

January 31, 2006

Abstract

Most chess-playing programs are based on well known algorithms and determinisitc evaluation functions. Due to the complexity of this game, it seems however that using complex systems such as neural networks or genetic algorithms may help improve (or at least compete with) the performance of said programs.

This seminar aims to present several projects and papers making use of these "alternative" methods to design chess programs, details about their implementations and their actual performance compared to "classic" chess algorithms or human players.

Contents

1 Introduction

Chess is known to be a complex game. However the rules of this game are quite simple, the complexity comes from the number of positions, and the number of moves a player can do (around 35 each turn).

When talking about computer chess players, one mostly thinks only about algorithms such as minimax. These algorithms associate a score to possible moves in a totally deterministic way, then choose the move which has the best score. Even if these are the most used algorithms, their implementation has some annoying drawbacks.

Another approach to designing computer chess players would be to *teach* the computer to become a good chess-player without hard coding its behavior. This might be possible by using techniques such as neural network or genetic algorithms.

In the following, some classical, determinitic approaches to chess-player design will be quickly explained, along with their drawbacks. Then focus will be given to several projects not using these techniques in favor of methods which try to have the computer learn by itself to become a better chess-player. The implementation of these will be described and their performance at playing chess detailed.

2 The minimax algorithm

Minimax (sometimes called min-max) is a common algorithm [1] which can be used for almost every two player board games. One important fact is that games usable with minimax are those where no information about the other player's possible moves is concealed (no secret piece or card, both player have the exact same knowledge about the game). It (or its variants) is the most commonly used in chess programs.

Minimax relies on a search tree of the attainable board configuration from the current situation (*i.e.* a tree of the possible moves, see fig. 1). It then searches through this tree for the best move to play.

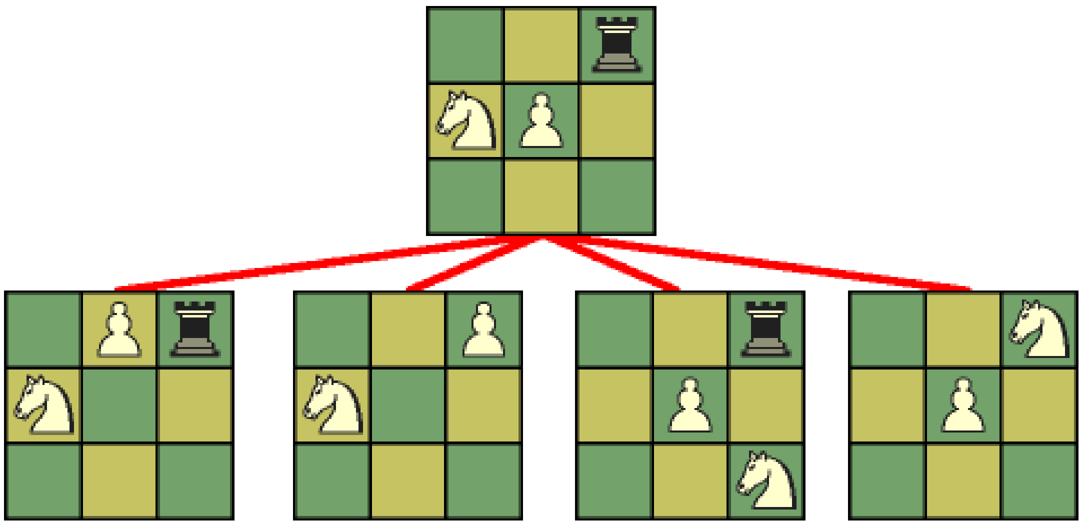

Figure 1: A one-degree depth minimax tree for a (very) simplified chess position.

The algorithm is based on the assumption that the opposing party always makes the best move (*i.e.* the worst move for the current player). This behavior is known as *perfect play*. It explores (depth-first), the possible next moves until the very end of the game.

A score is assigned to the leafs of the search tree depending on the outcome of the game (win, loss, draw). This score is then propagated up into the tree. Every level of the tree represents the possible moves for one of the players. As they have opposite objectives, their decision will not be the same; the current player will try to make the move maximizing the score, whereas his opponent will try to maximize his own evaluation of the game, thus minimizing the current player's score. Minimax is then an alternation of maximization and minimization of the score (see fig. 2).

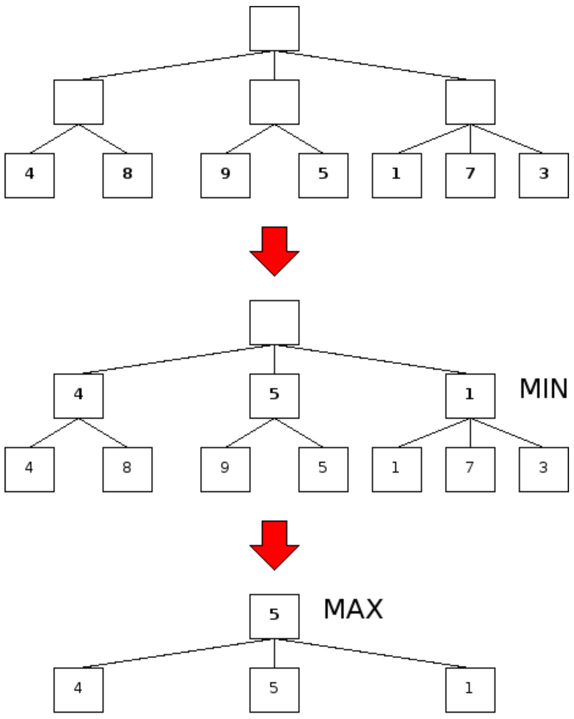

Figure 2: The propagation of the scores using the minimax algorithm for a relatively short search tree.

3 Classical approaches

3.1 Heuristic scoring

Due to the complexity of the chess game, it is not reasonnable to implement the minimax algorithm *as-is*. The computer time needed to explore all the possibilities would be too long.

To solve this problem, the tree is not explored until the end of the game, but only to a depth n. The position is then given a score according to evaluation functions considering the features of the board configuration.

3.1.1 Board evaluation

The usual score function takes into account several parameters of the pieces positions in order to compute the evaluation of the situation. This type of evaluation function is usually checking some or all of the following features [2].

material balance every piece can be given a value according to its power (*e.g.* a pawn will be given a smaller value than the queen), the relative power of both players can then be determined by comparing the value of their pieces;

number of possible moves the more possible moves a player can play, the less he is likely to be stuck in an uneasy situation, thus their number can be counted positively in the scoring function;

board control *i.e.* , the number of parts of the board where a player has more pieces than his opponent which can, then, be considered safe; the more control a player has on the board, the easier it will be for him to set up his pieces in order to attack;

development of the pieces is an important factor as the pieces are not useful if they do not move, whereas they can be used to gain a better control of the board or threaten some of the opponent's pieces;

pawns formation as they are numerous, the pawns can either be very useful or disturb the evolution of the game, then it is important to place them correctly, which makes their positions an important criterium to take into account while evaluating a position;

king safety the final goal when playing chess is to manage to threaten the opposing king so that it can't escape, considering how well one's king is protected and how easily one's piece can attack the opponent's king are good insights about how good the current board configuration is.

Defining functions evaluating these parameters is not straightforward, but the main problem is usually attributing a weight to all of these attributes in order to determine the score of a position. This is mostly donne by trial-and-error which may not be very efficient.

3.1.2 Drawbacks

Even if the minimax algorithm theoretically manages to win, as it tries to find the path to a winning position, some drawbacks appear in its implementation using this kind of evaluation function.

First, it is obvious that the greater the explored depth n will be, the longer it will take for the computer to determine what it should play. A trade-off has to be made between speed and exploration of the outcomes of the possible moves.

Another, most important, problem of the actual implementation of the minimax is called the *horizon effect*. As the algorithm only explores a depth of n turns, it cannot detect some possible very good (or very bad) positions, which may drastically change the score it would attribute to a given leaf, situated at level $n + 1$ or deeper.

3.1.3 Alpha/beta cutoff

An optimized version of the minimax algorithm is the alpha/beta algorithm [3]. It tries to minimize the exploration in the tree by cutting branches that will not give better results than what has been already explored (fig. 3 on the following page).

This algorithm usually decreases the computation time needed to choose between the possible moves. However, it still suffers from the horizon effects which it does not handle better than the minimax as the board evaluations do not usually take into account the possible next moves.

3.2 Plies databases

Current chess programs usually make use of databases in which interesting sequences of moves are stored. This can be seen as a way to give memory to the chess-playing computers. Usually, only openings (as on fig. 4 on the next page) and endgames (*i.e.* final periods of the game when only a few pieces remain on the board) patterns are kept in these databases.

These databases cannot be used totally alone. They can however provide information to chess-playing algorithms; they can give a good heuristic in order to score some positions which might be encountered during a minimax search.

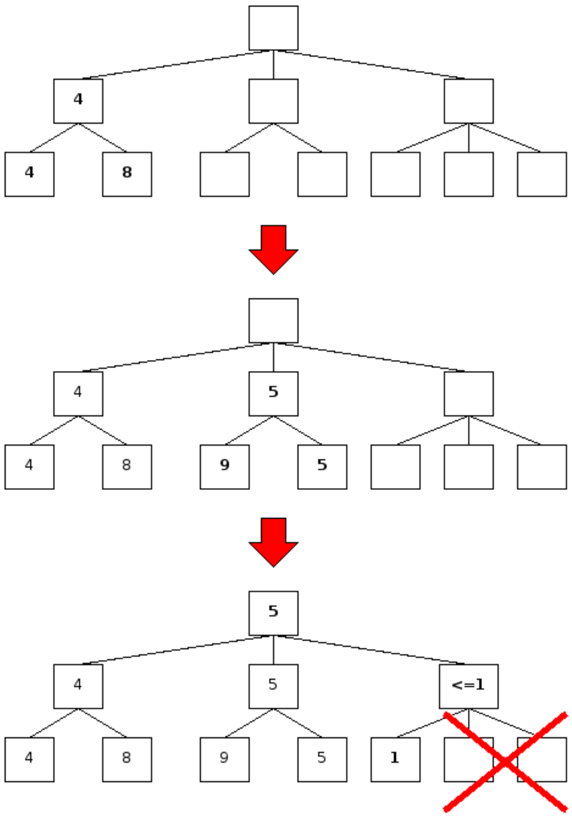

Figure 3: The alpha/beta method can be used to cut branches of the minimax search tree which will not give interesting scores.

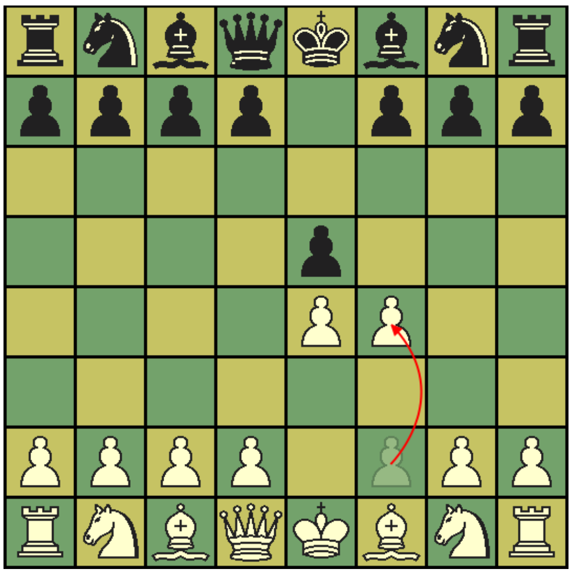

Figure 4: The King's Gambit is a well known opening where the player proposing the gambit is exchanging one of his pawns with control of the center of the board.

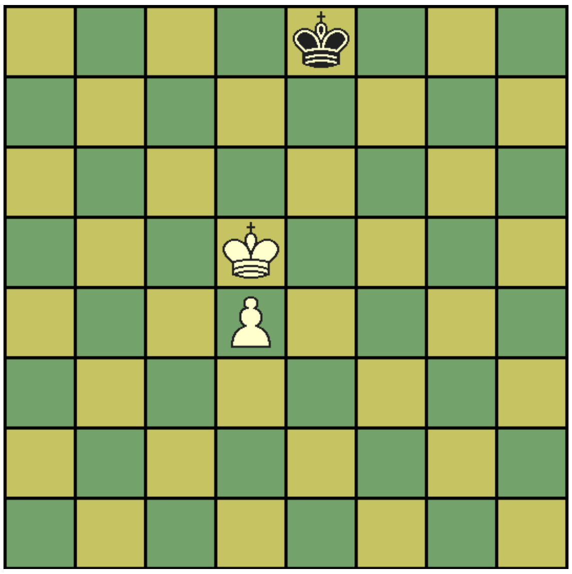

Figure 5: An example king-pawn endgame from [4].

However adding interesting strategic behavior to the games, the program using these databases only know a fixed number of hard-coded plies. The strategy that the computer player may adopt is then totally dependent on the human who actually filled these databases and cannot learn more of these plies.

4 Complex system based methods

As opposed to what has been seen above, the methods which are presented in the following try to make the computer algorithms evolve and learn by themselves or minimize the computation time needed, for example to look in a database of games.

Most of the following can give alternative board evaluation functions diminishing the horizon effect of the minimax and its derivatives. Some of them may even be used on their own (no search algorithm).

4.1 Genetically evolving the weights of the evaluation function

As part of a machine learning course final project [4], Chris Wyman developped a program to train computer chess players using genetic algorithms. This project has only focused on king-pawn endgames (fig. 5) configurations in order to reduce the complexity of the problems, but the method can easily be rescaled to the whole chess game.

4.1.1 Individuals

The individual is the set of weights that the evaluation function gives to several attributes of the board configuration. This is a good alternative to the trial-and-error way of determining those.

As the black and white players do not pursue the same goal (the white player will try to win, whereas the black one will try to draw the game), two set of weights have to be evolved for each player, depending on which side it is playing.

4.1.2 Fitness function

The evaluation of each individual is done by making them compete against each other. The outcome (win, loss, draw) of these matches and their expectedness (an unexpected loss is worse than an expected one) is used to compute the fitness value for a specific set of weights.

The formula for the fitness function, as given in the paper, is

$$\frac{W_{ew}N_{ew} + W_{uw}N_{uw} + W_{ged}N_{ged} + W_{bed}N_{bed} + W_{gud}N_{gud} + W_{bud}N_{bud} + W_{el}N_{el} + W_{ul}N_{ul}}{N},$$

where the W (parameters of the GA) are the weights given to each outcome, and the N are the number of games with the specific outcome ($e \rightarrow$ expected, $u \rightarrow$ unexpected, $w \rightarrow$ win, $d \rightarrow$ draw, $l \rightarrow$ lose, $g \rightarrow$ good, $b \rightarrow$ bad). N is the total number of games.

It is interesting to note that the draw outcomes are classified into two categories: good and bad. Once again, this is due to the fact that chess is not a symetric game where the whites have an advantage. Drawing while playing whites will be considered a bad thing, whereas drawing when playing blacks will be a good thing.

4.1.3 GA parameters and specificities

A somewhat standard GA has been used, using 4 bits variables for each weight in the chromosome (some attempts at using 8 bits have been made to no avail), with mutation.

The generation evolution scheme is a little more complex than usual. The most common method is to select two individuals according to their fitness and have them reproduce, the offsprings replacing their parents in the next generation. In the scheme used here, only the individuals with the highest fitnesses are allowed to reproduce with each other. The individuals with the lowest fitnesses die and are replaced by the newly created offsprings. Finally, all the still-alive individuals from the last generation are kept in the new one.

In order to avoid overfitting, a maximum of 21 initial board positions were used. For the computation time to be kept reasonnable, it has also been necessary to randomly select players for a game instead of having all the individuals competing with every other for every initial board.

An interesting *bootstrapping* method in order to reduce the computation time to have a population of fairly good players for both side has been used. It consists of a single hard-coded black player against which the whole population plays. This leads to a white player which can play perfectly against this hard-coded black player. This white player is then used to train a black player, and so on. Using this approach, and repeating the process three times, it has been possible to train an always-drawing (against a human player) black player, and an interesting white player (not always winning, but hard to defeat).

4.1.4 Performances of the trained players

The resulting trained players were good enough to challenge and sometimes defeat a human player.

It seems however, looking at the author's conclusions, that this project suffers a scaling problem, as the players should be trained for every possible endgame situation (more pawns, another remaining piece, ...) which might be very expensive in terms of computation time and memory to store the players.

4.2 Neural-network-based position evaluation

NeuroChess [5] is an attempt to develop board evaluation functions using a neural network. The neural network is trained using temporal differencing with an explanation-based learning. In both cases, the process consists of presenting complete games and their outcome to the learning algorithm, which will then update the weights of the neural networks appropriately. The board configurations are presented to the neural network as a high-dimensional (175) vector.

First trained from databases of masters' games as a bootstrap, NeuroChess is then able to compete against itself in order to learn more generic, or unexpected, situations.

4.2.1 Temporal differencing

The goal of this method is to train a neural network for it to be able to reproduce evaluations of boards from already played games. In the end, it will hopefully be able to evaluate any given board state and give accurate indications about what the next move should be.

The evaluation of the final board configuration is 0 if the game is drawn, 1 (*resp.* -1) if the game is won (*resp.* lost) by the currently trained player. The scores of the previous states of the game, leading to the final position, are then derived from this evaluation using the equation from [5],

$$V^{\text{target}}(s_t) = \gamma \cdot V(s_{t+2}),$$

where $V(s_t)$ is the evaluation of board state s_t, $\gamma < 1$ a *discount factor* and s_{t+2} the board state two plies later (*i.e.* after one move by the current player, and one by his opponent).

The discount factor is used to gradually decrease the evaluation of the boards as the learning studies older board configurations. This has the desired effects of getting the evaluation of the board close to 0, that is, a fair game, as the pieces get back to there initial situation, and to give higher evaluations to configurations closest to a winning move, which favors quick games.

4.2.2 Explanation-based learning

Explanation-based learning (EBL) is used to reduce the training time of neural networks. Instead of only training the networks with observations from the board, information about the actual rules of chess are also given to the training algorithm in order to help it find the relevant features of given board position.

The rules of chess are "explained" to the EBL algorithm using another neural network M. This *chess model* is trained (using back-propagation) by being presented boards s_t and s_{t+2} from a database of chess games.

Using this approach, not only $V(s_t)$ but also its slope $\frac{\partial V(s_t)}{\partial s_t}$ can be determined by the neural network. This information can then be used in order to determine which of the features of the board should be change in order to increase quickly the score of the position.

Using this technique, it seems that algorithms such as the minimax may no longer be necessary. In effect, the evaluation function can not only evaluate a bord configuration, but also give, through this slope, the best direction in feature space (*i.e.* the best move) to play.

4.2.3 Performances

The program has been trained both using grand masters' games database and by playing against itself. It has then been compared to GNU Chess's chess engine, which is using an alpha/beta search, a "regular" board evaluation function and an opening book.

NeuroChess performs quite badly at playing against this deterministic engine, losing in more that 70% of the games after having been trained over more than 2000 games. This figure, however, has been ever-diminishing since the begining of the training. One might expect that, with a sufficiently long training, NeuroChess may be able to outperform GNU Chess. It is also important to note that, compared to a regular backprop-taught neural network, the EBL one manages to win more games more rapidly.

4.3 Genetically programming a maximum search depth function

Using search functions such as minimax or its variants, one of the main issue is determining the depth to wich the search should go down. This is usually worked around by hardcoding a maximum search-depth in the algorithm. A really interesting solution to this problem would be to be able to predict at which depth a potential winning move may happen. It would then be pointless to lose more time searching deeper.

Genetic programming has been tested with some success [6] to a subset of chess, namely the King-Rook-King endgame (fig. 6 on the next page). Two types of functions have been evolved, solving either the *grand KRK problem* (determining the depth at which a checkmate happens) or the *petite KRK problem* (classifying games depending on whether or not they lead to a checkmate in exactly the specified number of moves).

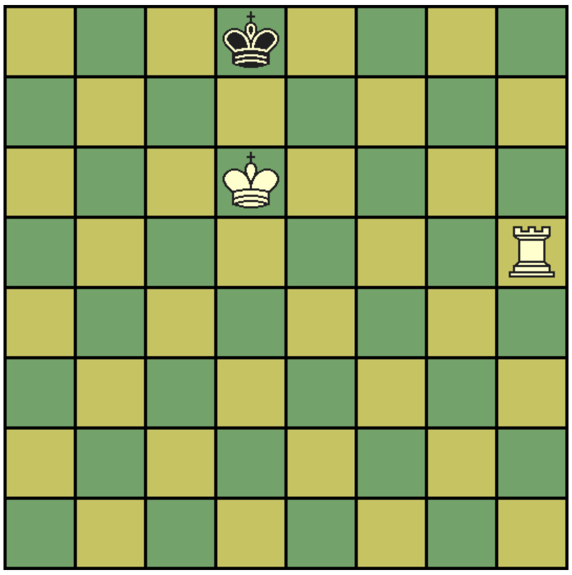

Figure 6: An example KRK endgame; black's turn, white wins in 3 moves.

4.3.1 Terminals and functions

Genetic programming is a genetic algorithm manipulating trees. These trees represent the function being evolved. The branches of the tree are some basic functions whereas the leafs are terminals, actual values that are to be used in order to compute the function's output.

In the KRK-endgame, there are mostly 6 parameters describing the game situation. These are used as terminals:

- wk_r, wk_f, the white king's rank (column) and file (line);
- wr_r, wr_f, the white rook's rank and file;
- bk_r, bk_f, the black king's rank and file.

However not mentionned in the paper, it seems necessary to have two constant terminals, such as 0 and 1 or 1 and 2. All the terminals are integers.

The functions used to construct the objective are very basic and not at all, expect for one, related to chess:

- $edge(i)$ which returns 1 or 2 depending on whether i is an edge of the board (*i.e.* $i = 1, 8$);
- $distance(i, j)$ which returns the (positive) difference between i and j;
- $ifthen(i, j, k)$ which returns j or k depending if $i = 1$ or not;
- $compare(i, j)$ which returns 1 is i is lesser than j, and 2 otherwise.

4.3.2 Fitness functions

The endgame configurations given to the algorithm come from a database of KRK endgames, in which are also stored the numbers, lesser or equal to 16, of moves leading to a checkmate.

It is then possible to compare, in the case of the grand KRK, the output value of the evolved functions with this number to determine if it performed correctly. The grand KRK fitness function is then computed as

$$F = 1 - \frac{\sum_0^{16} \frac{C_i - I_i}{N_i}}{17}.$$

It compares, for each possible number of moves i, the number C_i of endgames correctly found to finish in i moves with the number I_i of incorrect outputs. The perfectly fitted individual will get a fitness of $F = 0$ whereas the worse one will get fitnesses $F > 0$.

For the petite KRK problem (actually there are 16 of these functions to evolve, one for each class), which is a classifying problem, the fitness function is simpler to understand:

$$F = 1 + T_p - (C_p - I_p) - \frac{C_n}{T_n}.$$

Where the T's are the total number of positive (p) or negative (n) classifications, the C's are the correct classifications and the I's are the erroneous ones. This fitness function will be 1 for a perfectly classifying function, and greater than that for worse ones.

4.3.3 GA parameters

The *jrgp* genetic programming system has been used to evolve the individuals. Both problems were evolved using a crossover probability of 75% and a mutation probability of 15%. The population consited of 5000 individuals for the grand KRK, but only 1000 for the petite one, they were trained for all the positions given in the KRK database.

4.3.4 Results

In the end, the GA produced a grand KRK function giving the right depth in only 41% of the cases (after 32 generations). The petite KRK function (only one was actually completed) performed much better, with 97% correct classification after 145 generations.

It may then seem interesting to use the petite KRK functions in a potential maximum-depth searching function, but one has to keep in mind that there are sixteen of these to evaluate for only one grand KRK function, which may be much more time consuming.

One should also be attentive to the fact that the individuals were trained over the whole database, and the risk of having an overfitted function, not giving correct results for other endgames, is high.

4.4 Next-move-deciding neural network

The distributed chess project [7] aims at “playing chess by pattern recognition”. The goal of the project is to genetically evolve a neural network able to recognize board configurations and determine the next interesting moves appropriately. However too few information about the actual implementation are given, this project is interesting for several reasons.

The first interesting feature is that is does *not* make use of minimax-derived algorithms. Contrary to the preceding neural network, this one does not evaluate a given position, but rather “decides” what the next move will be played.

Another characteristic which is worth noticing: the way the genetic algorithm is run. Instead of running the algorithm on one single computer or little park of workstations, but on the whole internet. It proposes people with idle computers or spare computing time to run the GA. Besides speeding the execution of the genetic algorithms, this has the interesting effect of creating subpopulations, which is usually a good thing to have in order to avoid early convergence.

The neural networks are not trained on a complete game. They are proposed a number of games which are supposed to focus on some specific features or periods of the game (endgame, ...).

The final results on the training and test sets have shown that the neural networks evolved using this method actually managed to perform better (depending on the training set). This tends to show that it might be possible to dig further away in order to evolve better chess-playing neural networks.

Using this kind of neural-network could also be useful in conjonction with more standard search algorithms like alpha/beta. Valuable information such as the most interesting moves could be used to reorder the explored node and cut earlier, thus increasing the search speed of such algorithms by allowing them to cut branches earlier.

5 Conclusion

Some of the results obtained by complex systems-based approaches are interesting, in all cases, the training or evolution produced better chess player than at the beginning. However, these chess players were either not able to play a complete game because they only focused on some specific period of the game or sub-function, or they were not good enough to fairly compete with other, deterministic, chess playing programs.

In most cases, the limiting factor is the long training time needed to end up with a hopefully good versatile player.

The approaches presented above, even if they do not seem good enough to be used on their own in chess applications, maybe be integrated into already existing programs in order to improve the heurisitics, evaluation functions,...

One can, anyway, hope to see these methods being more widely used, in addition to more classic chess algorithm, in order to build really interesting chess players which may be able to come up with some new, previously unthought of, tactics.

References

[1] Paulo Pinto. Minimax explained. http://ai-depot.com/LogicGames/MiniMax.html.

[2] François Dominic Laramée. Chess programming part vi: Evaluation functions. http://www.gamedev.net/reference/articles/article1208.asp.

[3] Paulo Pinto. Optimisations (for minimax). http://ai-depot.com/LogicGames/MiniMax-Optimisations.html.

[4] Chris Wyman. Using genetic algorithms to learn weights for simple king-pawn chess endgames, 1999. http://www.cs.utah.edu/~wyman/classes/ML_proj/paper.html.

[5] Sebastian Thrun. Learning to play the game of chess. In G. Tesauro, D. Touretzky, and T. Leen, editors, *Advances in Neural Information Processing Systems 7*, pages 1069–1076. The MIT Press, Cambridge, MA, 1995. http://citeseer.ist.psu.edu/article/thrun95learning.html.

[6] David Gleich. Machine learning in computer chess: Genetic programming and KRK, 2003. http://citeseer.ist.psu.edu/gleich03machine.html.

[7] Ralf Seliger. The distributed chess project. http://neural-chess.netfirms.com/HTML/project.html.