Chess AI improvement through an evolutionary approach

Cédric Guillot, University of Calgary (cpguillo@ucalgary.ca)

April 30, 2013

Abstract

This paper presents the results in the attempt of improving a classic chess AI by using an evolutionary approach. The general idea is to fine-tune the parameters used by the chess AI (pieces values) in its board evaluation.

1 Introduction and Goals

Chess AI has been a subject of interest for over 70 years now. The first paper program that could play a game of chess was written in 1951 by Alan turing, and the most symbolic achievement was Deep Blue (chess AI developed by IBM) that beated the former world champion Garry Kasparov in a regular cheess match.
Strategies for chess AI have improved but they still use parameters assigned to each kind of pieces in order to evaluate how favorable the chess board currently is to us. For that reason, this project aims at fine-tuning these parameters, more specifically the values of pawn, knight, bishop, rook and queen (the king being invaluable anyway). Using an evolutionary algorithm will allow close to optimal values (for this particular strategy) to emerge and so provide a more difficult AI to beat by a human player.

2 Implementation

The first step for applying the algorithm was to find a chess engine that would provide the board environment and the search and evalution functions that make the AI as such. It was widely influenced by the choice of a performance programming language, here the C language.

2.1 Tools

Tom Kerrigan's Simple Chess Program (TSCP) was chosen because it is simple enough while at the same time providing every components for playing a full game of chess. The emphasis of this engine is put on the documentation so that one can quickly start working with it. Its AI is based on the standard alpha-beta search algorithm.

The second tool that was used is GNU Xboard, as it provided a graphical interface for playing chess and compatible with TSCP. It helped with both visualising the way that the TSCP AI played and playing games against the different AIs that were generated (through finding better pieces values).

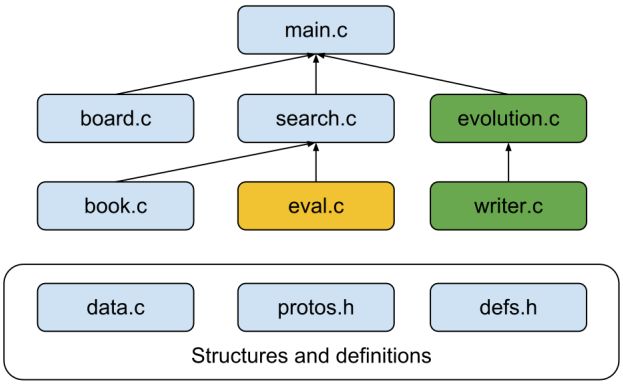

Figure 1: TSCP and evolution algorithm plugin architecture. In green, the evolutionary plugin. In yellow, the evaluation file that is modified to change the values used by the AI.

2.2 Project architecture

Implementing the evolutionary strategy for TSCP was done with a plugin philosophy in mind. The following chart shows the architecture once the evolutionary plugin has been added:

Basically, main.c still controls the flow of the program, but instead of asking for a move to the player, it will ask the evolution.c module for the next game to play by a call to next_game() method. The writer module takes care of keeping an history file with the best individuals from each generation. It helps with drawing the pieces values charts that we focus on in the results section.

2.3 Algorithm

Evolving the values used for the 5 types of pieces that we are interested in is done using an special kind of evolutionary algorithm. We call 'individual' for that algorithm the set of 5 values for pawn, knight, bishop, rook and queen (initial values of respectively 100, 300, 300, 500, 900).

2.3.1 Evolution strategy

Before we get into details, it is crucial to understand that the values we are evolving remain the same during

a game between two individuals (the BOARD evaluation function uses the same values from the individuals during the same game). Our OWN evaluation function is based on the games that individuals play against each other (with their values feeded into the chess AI for the BOARD evaluation function).
The algorithm is described below:

- The μ parents breed λ children,
- Children are mutated randomly,
- Parents and children each play two games against one another (once as black, once as white),
- Depending on the number of points collected, the μ better individuals are selected as the new parents for the next generation.

As there are $\mu+\lambda$ individuals per generation, it amounts to $\frac{(\mu+\lambda)\times(\mu+\lambda-1)}{2}$ games to play for each generation. This is the main reason why we decided to choose the performance of the C language.
The points distributed after one game are as follow:

- Victory: 3 points,
- Defeat: 0 point,
- Stalemate, draw (by repetition or maximum number of moves (50) reached): 1 point.

Hence, our evaluation function is relative to the individuals that are part of the generation. However, as we always select the best individuals between parents and children ($\mu + \lambda$ strategy), our approach does not suffer from degeneration of individuals because of the relative evaluation function.

2.3.2 Boundary checking parameters

For reasons of convenience, we decided to implement a boundary checking mechanism to ensure that the values remain reasonable throughout the evolution. That prevented some values to get ridiculously far away from their expected range. The ranges that were used are described below:

- Pawn: $10 < value < 200$,
- Knight: $200 < value < 400$,
- Bishop: $200 < value < 400$,
- Rook: $400 < value < 600$,
- Queen: $700 < value < 1100$.

2.3.3 Strategy parameters

Initially, the random change that was applied to each child was set to be a random integer between -15 and 15. Then, due to the ranges of the values, the variations got changed to random number in [-5;5], [-8;8], [-13;13], [-15;15], [-20;20] respectively for pawn, knight, bishop, rook and queen. Finally, we tried to implement Rechenberg's heuristics for modifying the intervals.

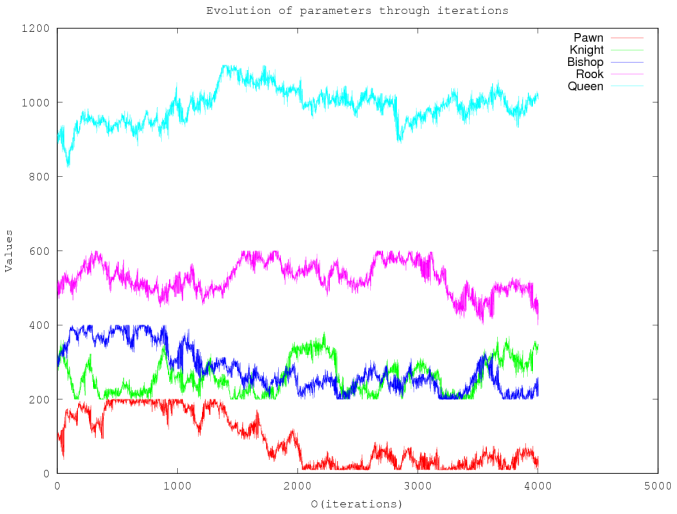

Figure 2: Values evolution through iterations (fixed intervals)

3 Results and discussion

We first describe the experimental parameters before presenting the results (by interpreting the graphs given by the evolution of individuals).

3.1 Experimental parameters

The depth of search for the experiment is $n = 1$. This value was chosen in order to test the former and new AI by a person with a beginner level. The initial population was made of 4 identical individuals with values 100, 300, 300, 500, 900. The evolution parameters were set to $\mu = 4$ and $\lambda = 2\mu = 8$. The algorithm was left running for 1000 or 500 iterations (a history of individuals of around 4000 or 2000).

3.2 Evolved AI

Two experiments were conducted: once involved fixed intervals for children mutation while the other one implemented the Rechenberg's heuristics. It is to be noted that for practical reasons, the two simulations were not held over the same number of iterations, although they are of the same order of magnitude (1000 and 500 approximately). Both are presented in figure 2 and 3:

As we can see, Rechenberg's heuristics is not applicable in this context, as the individuals are changing way too fast in the simulation. This can be explained by the huge mutations that take place because the intervals are too important. The most interesting experiment is with fixed intervals. However, it is hard to be satisfied with the results from above. First of all, there is no convergence towards any specific value for any of the 5 values. Secondly, the boundaries applied on the values may have impaired evolutions that might have occured.

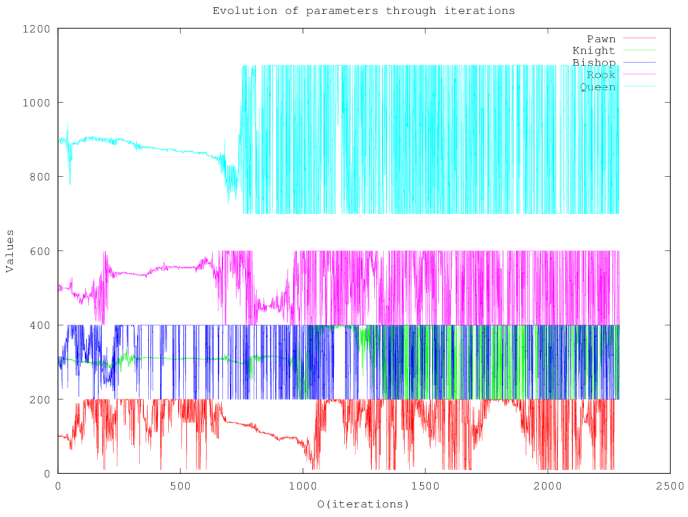

Figure 3: Values evolution through iterations (Rechenberg's heuristics)

But we can still see 3 things emerging. Firstly, the pawn value may be lower than expected (on average lower than 100). Secondly, the queen value is slightly higher than 900, which confirms than loosing your queen in a chess game means a great loss for the win. Finally, the knight seems to be superior to the bishop, which means practically means that trading your bishop for the opponent's knight is a good move.

3.3 Discussion

The study that was lead on improving the values suffers from a couple of shortcomings. At the end of the 'fixed intervals' simulation, both AI played against one another, and both games (one as black then white for both AIs) ended up on a draw. That means that no significant improvement can be seen with the method chosen for comparing the AIs.

The first problem that could be responsible for the lack of improvement is the depth of the search tree that was reduced to $n = 1$. The AIs could not properly exercise the values at a reasonable level.
The major problem was due to the relative fitness function that was introduced. Even if we kept the fittest individuals after each generation, former versions of the program ended up with worse solutions than the initial ones. Further study would be required to pinpoint at the phenomenon happening. The solution that should be explored should involve an absolute ranking system; the ELO ratings adapted for AIs would provide the natural rankings for rating the individuals playing the chess games.

3.4 Conclusion

In conclusion, this project aimed at improving the chess AI of a standard chess engine. Although it failed at providing a strictly better set of values, interesting trends were observed to come up with an equivalently good AI. This could serve in the future as a starting point for improving a more general set of chess AIs with different built-in strategies.

References

[1] Hallam Nasreddine, Hendra Suhanto Poh and Graham Kendall : Using an Evolutionary Algorithm for the Tuning of a Chess Evaluation Function Based on a Dynamic Boundary Strategy
http://red.cs.nott.ac.uk/~gxk/papers/ieeecis2006.pdf

[2] DAVID B. FOGEL, FELLOW, IEEE, TIMOTHY J. HAYS, SARAH L. HAHN, AND JAMES QUON : A Self-Learning Evolutionary Chess Program
http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=1

[3] Graham Kendall, Glenn Whitwell : An Evolutionary Approach for the Tuning of a Chess Evaluation Function using Population Dynamics
http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=9

[4] Crafty : http://www.craftychess.com/