**BABES-BOLYAI UNIVERSITY CLUJ-NAPOCA**

**FACULTY OF MATHEMATICA AND COMPUTER SCIENCE**

**SPECIALIZATION COMPUTER SCIENCE**

**DIPLOMA THESIS**

Applying chess bot optimizations in various variants and evaluating their performance impact

**Coordinator:** lect. Cojocar Dan

**Author:** Craciun Ioan-Flaviu

**2020**

Table of Contents

1. Introduction1
2. State of the art in chess engines 2
   1. AlphaZero 3
   2. Stockfish4
   3. LeelaChess5
3. Chess algorithm optimizaitons
   1. Alpha-Beta pruning
   2. Iterative Deepening
   3. Inverse Petrosian evaluation
   4. Quiescence Search
   5. Chess-specific optimizations
4. Chess engine implementation6
   1. Specifications7
   2. Optimizations used8
   3. Optimization impact in Elo evaluation8
5. Performance Evaluation9
   1. Bot vs Low-skilled human10
   2. ELO rating approximation11
6. Applying the algorithm to other games12
   1. Checkers13
   2. Go14
   3. Fischer Chess14
7. Conclusions15

Bibliography16

1. **Introduction**

Chess is a complex game with many possible moves at each turn, attempting to compute the best move is difficult since the number of different games hits millions only a few moves in, growing exponentially afterwards. Humans will discard the obviously wrong moves, but still wind up only building a shallow game tree in their minds, only the very top minds being able to apply very good heuristics to increase the depth to nearly double.

Here is where computers come in, they do not need perfect heuristics, they can just build a massive game tree and through a min-max algorithm with good pruning techniques it can easily outperform intermediate level players. Another very important advantage a computer has over human minds is parallelizability[1], a human has only one brain, but a computer can have tens or hundreds of cpus that can aid computation.

In this thesis we employ several methods and compare their efficacy in developing an AI capable of defeating average humans. Besides pruning and parallelization we will also analyze heuristics such as quiescence search[2], different evaluation functions and inverse petrosian evaluation[3].

1. **State of the art in chess engines**

This chapter describes the current best implementations of the problem we set out to solve. Although there are many chess algorithms out there, I have chosen those 3 that I think are the most important and relevant to current day research

* 1. **Elo Rating**

The Elo Rating system was created by Arpad Elo in the mid twentieth century to gauge relative performance in zero-sum games like chess. It is designed in such a way that it is able to predict the outcome of a game between two parties. For example a player whose rating is 100 points above their opponent is expected to score 64% of points, while for a 200 difference the expectation rises to 76%.

It is a logarithmic scale, meaning a 100 points difference predicts the same outcome regardless of the score of the two players, be it 1100 and 1200, or 2400 and 2500. In fact the precise expected score of player A where player A has rating RA and player B has rating RB is . Similarly, for player B the expected rating is .

Supposing player A achieves SA points, the new rating of player A will be where K Is tipically 16 for masters and 32 for weaker players[4].

* 1. **AlphaZero**

In 2017 the research company DeepMind unveiled AlphaZero, a ground-breaking neural network algorithm that after 4 hours of training matched the best algorithm to that date, Stockfish, and in 9 hours of training it was able to defeat Stockfish in 28 games out of 100, drawing the remainder.

It is important because it is among the first neural network based engines to achieve high performance. While the classical alpha-beta algorithms make heavy use of chess specific heuristics and observations, AlphaZero knows absolutely nothing but the basic rules of chess and simply plays the game by itself until sufficiently trained, at which point it can outsmart even Stockfish.

AlphaZero is also able to play Go and Shogi, and can be generalized to play Atari and board games, which is even more impressive that such a generally applicable algorithm can master so many games.

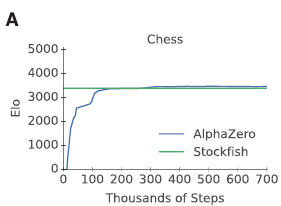


Fig. 1: Evolution of Elo rating for AlphaZero [5]

As per figure 1 we can see that the AlphaZero algorithm achieves an Elo rating above Stockfish very quickly, and settles around 100 or so more points than Stockfish. The results can not be extrapolated as being valid in the context of other engines as well since the study restricted its comparison to just Stockfish, but it is still very significant to consistently defeat the top performing engine in the field [5].

* 1. **Stockfish**

Stockfish is one of the strongest alpha-beta search algorithms out there, and also the best CPU algorithm (AlphaZero requires a GPU for training). This is the most one can currently hope to achieve with the classical backtracking algorithm and for years ruled uncontested until AlphaZero came onto the scene. Additionally, it is free and open source and it receives many contributions from community developers.

In 2020 a neural network version had been released and the developers claim a strong improvement, but given that that came after the success of AlphaZero the engine is still mostly thought of as a very strong backtracking algorithm.

Stockfish is also regarded as the highest ranking algorithm for a long time due to its success in TCEC (Top Chess Engine Championship), where it won most of the cups and seasons, but soon might be dethroned due to the emergence of Leela Chess.

In fact, Stockfish is so widely regarded as the best publicly available algorithm that it’s used by nearly everyone, most notably Lichess as the tool that evaluates human moves and gives a very good idea of a player’s quality, even among grandmasters[6].

* 1. **Leela Chess Zero**

Following the success of AlphaZero and the publication of DeepMind’s paper, the chess community set out to implement its own “AlphaZero” and came up with Leela Chess. After playing 500 million games by itself it achieves performance that is comparable to that of Stockfish. Leela does not however run on a super computer like AlphaZero, but relies on the community to run games locally and improve itself from self play among many volunteers.

In April 2018, just a few months after its release, Leela Chess has made the first neural network appearance at TCEC where it performed poorly, scoring only one victory, 2 draws and 25 losses. However it improved quickly and by February 2019 it had already brought home its first cup, not losing a single game the entire tournament.

Considering Leela Chess is attempting to replicate the success of AlphaZero, it is free and open source and the fact that it is still improving, it all makes this algorithm the most exciting to watch progress and the hope it that will eventually surpass all current implementations and become the number one chess engine in the world[7].

Leela Chess has a more rigorous Elo rating estimation than AlphaZero, as we can notice in figure 2 we have several estimates but they place the engine at around 3500 points at the time of writing. This is indeed comparable to Stockfish but it looks like it might still get better, so we can indeed hope that neural network take the king crown of chess engines.

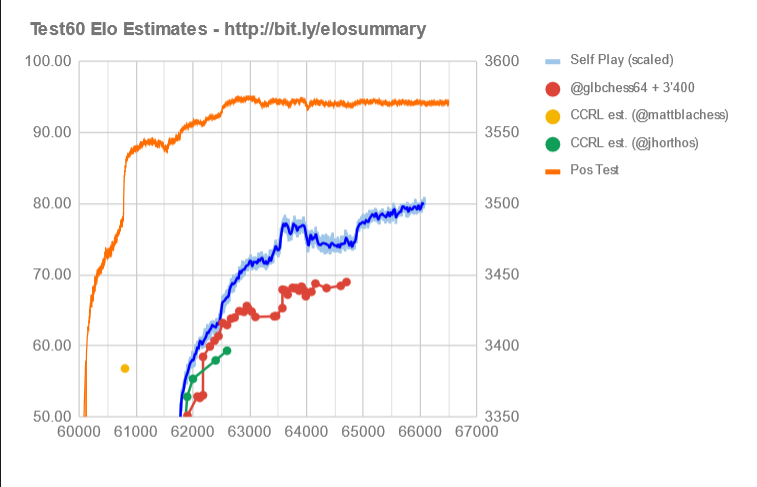


Fig. 2: The evolution of Leela’s Elo by various metrics (28-03-2022) [8]

1. **Chess Algorithm Optimizations**

In this chapter there will be a few performance enhancing ideas we can apply to the chess engine.

* 1. **Alpha-Beta Pruning**

Alpha-Beta pruning is the practice of not considering nodes in the game tree that irrespective of their yield, they can no longer influence the value of the root node.

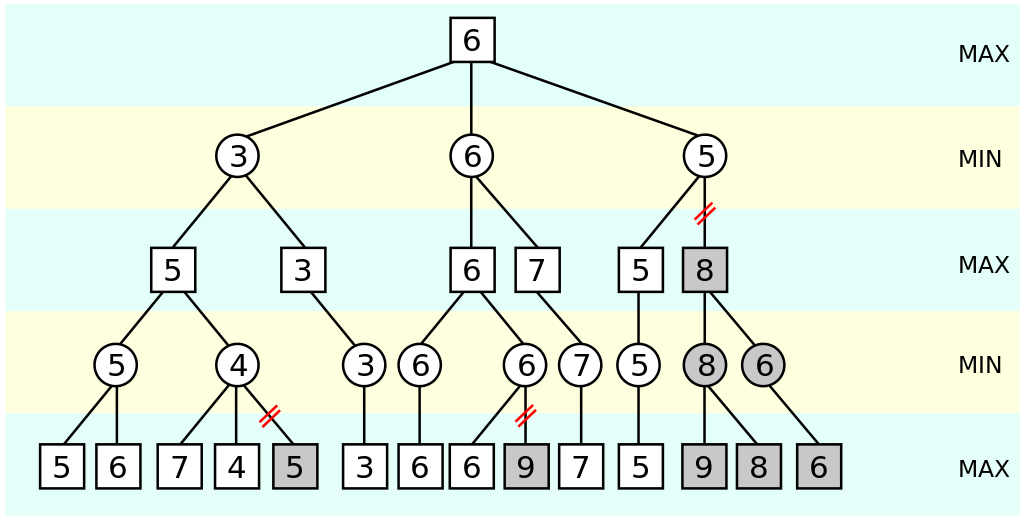


Fig 3.1.1: Alpa-Beta Pruning demonstration

Notice the grey nodes do not need to be calculated any more because for example the node with value 8 that is 2 levels below the root will not bring its parent to a value lower than 5, which will leave 6 as the maximum on the level below the root.

The pruning will optimize the search tree by quite a lot, thus will allow the search to get even deeper.

* 1. **Iterative deepening**

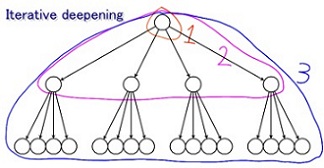


Fig 3.2.1: Iterative deepening example

Iterative deepening is the standard approach in writing a min-max algorithm. Instead of doing a fixed size search or anything else it is recommended to start with a depth 1 search, then a depth 2, and so on until you reach the time limit allocated to the move.

This is not as time consuming as it sounds, consider the fact that from each node you will usually have several options to pick from. This results in a game tree that is very wide, and the amount of nodes in the last level far surpasses the amount of nodes in all the other nodes combined.

* 1. **Inverse Petrosian evaluation**

Inverse Petrosian evaluation notices that overprotection, namely placing your pieces in such a position that there are more protections than necessary will strip your opponent of tactical strikes, thus he won’t find as many unusual moves than in the case as you not considering this. This idea is of course attributed to Tigran Petrosian as one of his great contributions.

The exact impact of the heuristic can be implemented as substracting times some constant from the evaluation of the position. This will promote positions with less defence between pieces. This will cause the computer to pick up on tactics more easily than a human and they will waste time looking for them.

* 1. **Quiescence Search**

This technique proposes we do not stop evaluating the game tree at a fixed depth, since it puts us in danger of leaving a position which has an obvious winning shot for the opponent, but that currently believes we have as many pieces as to feel safe.

A way to implement this is when we reached the end of the desired depth to also check whether there are captures available or checks we can throw. As long as this is still the case keep computing, otherwise finish the search and return evaluation.

* 1. **Chess specific heuristics**

The heuristics will be about which moves to consider first such as not moving kings apart from castling early in the game, giving greater weight to pawn, bishop and knight moves early in the game. A matrix of weights per piece can also be considered depending on their position on the board, a pawn close to promoting is more valuable than one on its home rank. Connected pawns are more valuable, connected passed pawns even more so, isolated and doubled pawns not as much. Castling is highly desirable, advancing pieces off their home ranks and close to the middle of the board is also desirable. Another thing to do is to give more weight to knights than bishops when there are many pieces on the board, around +30% in the early game and -30% in late game. Great weight should be given to moves that capture a valuable piece with a less valuable piece, moderate weight to moves that capture equal(-ish) pieces.

1. **Chess engine implementation**

In this chapter we discuss the implementation of the application and the technologies used in achieving it.

* 1. **Specification**

The project will serve as a way for humans to face the chess engine developed so far, you will be able to play against it and see if you can defeat it. This interface will also be used to emulate real games in the scope of determining the elo rating of the engine.

It will be separated between a frontend application, based on angular and a backend application based on Kotlin with Spring Boot. This is so that potentially multiple clients can be written, such as an android application, but the code for computation won’t be duplicated. Also the frontend code can run on a slow device and the backend can be run on a fast computer.

Another benefit to this separation is the ability to allow two such engines to play against each other, this is useful in determining whether certain optimizations improve the outcomes of the algorithm. Of course, we can simulate the elo rating in those cases as well, but it will be interesting to pit the two versions against each other and observe the results.

The final version of the application will have to make use of all of the optimizations, which are the following:

1. Alpha-Beta Pruning
2. Iterative Deepening
3. Inverse-Petrosian Evaluation
4. Quiescence search
5. Chess specific heuristics (different weights per piece, etc)

We aim to build an algorithm that will perform at least 1500 on lichess, that is the middle and starting point on the platform. This will indicate that it is at least as good as half of the human player base, which is an intermediate level.

The user interface will be simple, as the focus is the computation of moves, but it will allow users to move pieces on the table in accordance to the rules of chess. Besides the well known moves (rook moves orthogonally, bishop diagonally), the possible moves will include:

1. Castling (and lack of rights to castle)
2. En-passant
3. Promoting

These are the moves which require more attention during implementation, as they have complicated preconditions and postconditions. The algorithm will correctly assess the state of the game (win, loss, draw due to lack of moves, impossibility of checkmate from either side, 50 move without capture limit) and report it as such.

The desired outcome is a backend that efficiently processes the moves and uses little RAM and makes use of as many CPUs as possible.

**6. Bibliography**

[1] Guidry, C.M.M. and McClendon, C., 2009. Techniques to parallelize chess.

[2] Marrero Rodríguez, R., 2022. Developing a chess engine (Doctoral dissertation, Universitat Politècnica de València). Pg. 24

[3] Dechter, R., Flerova, N., Isbell, A., Pestano, C. and Ma, D., 2010. Rapier: A Chess Engine. Pg.4

[4] Elo Rating System - https://en.wikipedia.org/wiki/Elo\_rating\_system

[5] AlphaZero - <https://en.wikipedia.org/wiki/AlphaZero>

[6] Stockfish - <https://en.wikipedia.org/wiki/Stockfish>

[7] Leela Chess Zero - <https://en.wikipedia.org/wiki/Leela_Chess_Zero>

[8] Leela Chess Official Website - https://training.lczero.org/?full\_elo=1