**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

**2022**

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

*Abstract*

The aim of this project is to develop a chess engine based on the minimax approach, employ various optimizations and compare the performance of the resulting engine. The engine code will be written in Kotlin, a high-level object-oriented language, the features of which can be taken advantage of in order to easily modularize the code, reuse logic, follow the control flow and test and experiment easily. The frontend of the project will be an angular application that enables a player to easily play against the engine, but the backend code is used in more than just the frontend, namely making the engine play against other iterations of it and exploring the win rates.

1. **Introduction**

Chess is one of the oldest games invented by man still played today. It has its roots in the 15th century, yet it was only in 1997 that we invented machines that were able to outplay the best players in the world (the year DeepBlue beat reigning world champion Garry Kasparov). Naturally, over time chess engines have only gotten better, in fact so good that even evaluating the top performers is a daunting task, as there is no human benchmark anywhere near their level. One reason the engines have gotten so good over time is that the game is played between just two players, involving no hidden information or randomness. This makes chess a prime opportunity for computers to tackle playing, the reason being that we can employ more rigorous techniques than heuristics, or even combine the two in order to achieve peak performance. Indeed, many ideas, optimizations, tricks and heuristics have been developed during these last few decades, ideas we mean to put in practice and observe the results of. Engines are typically written in a fast low-level language such as C++, as such this thesis will also serve as an exploration into the challenges and benefits of using an object-oriented language, namely Kotlin.

The engine does not need perfect heuristics, it 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].

The human player will be able to play against the engine through an angular-driven web application, one which communicates with the engine via the Spring Boot backend that contains the methods provided by the engine. The optimizations will be described and detailed in the second chapter, each of which are implemented by the engine. The third chapter will describe the practical part of the thesis, the structure of the frontend side and for the backend. The fourth chapter describes the observed performance of various iterations of the algorithm, analysis of games played against it and lastly patterns observed in the engine’s play. Finally, the conclusions of the thesis are drawn, reiterating the key take-aways and the meaning behind results.

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.

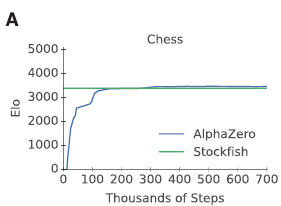


Fig. 1: Evolution of Elo rating for AlphaZero [5]

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.

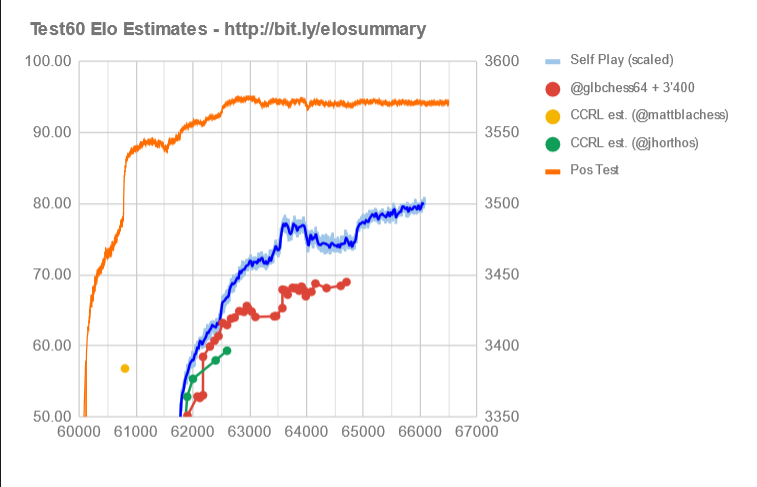


Fig. 2: The evolution of Leela’s Elo by various metrics (28-03-2022) [8]

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.

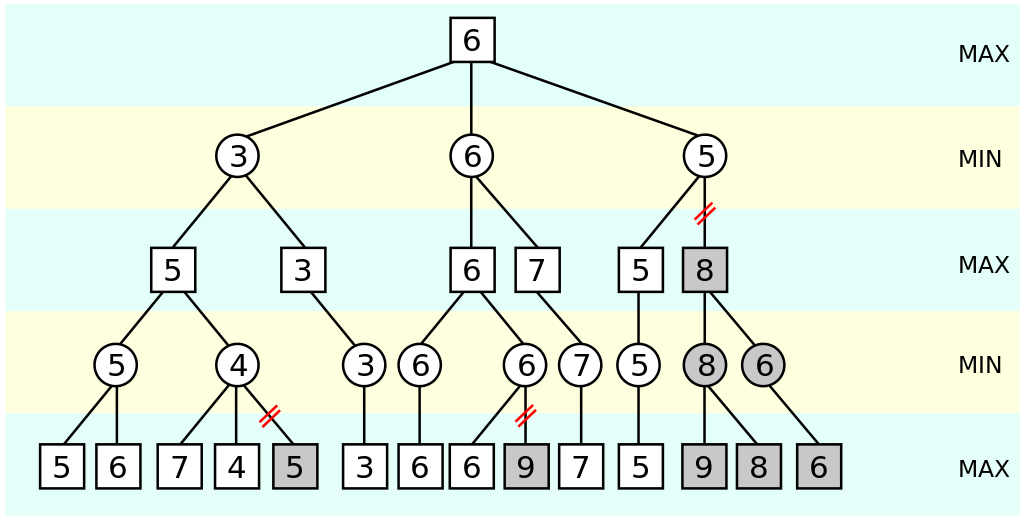


Fig 3.1.1: Alpa-Beta Pruning demonstration

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**

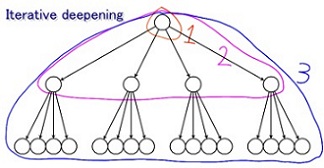


Fig 3.2.1: Iterative deepening example

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.

* 1. **Model**

The model of the application is a very important part of the implementation, as chess has many unexpected moves you have to consider, which complicate a natural OOP approach one would be inclined to take. Most moves will be piece A goes to position P, and possibly eliminate piece B. But you can castle (queenside or kingside) which involves moving two pieces at once. You can also promote a pawn which needs additional input about which piece to promote into. And as if it were not enough, the history of moves is also important due to it determining castling rights and en-passant availability.

Nonetheless, it has been accomplished and OOP has proven to be effective in organizing code, reducing code size, improving readability and fast tracking development.

* + 1. Moves

Moves are organized as classes that implement a simple interface. The base Move interface only contains an initial position of the piece moved and the color of the piece moved.

* BasicMove
  + The most common move you will make, it covers any type of move that takes a piece from position A to position B. An example is Rook from d1 to f1, or e2 pawn takes piece on f3.
  + As an additional field to the interface you will find final position
* CastlingMove
  + This move simply contains the castling type (queenside vs kingside) which together with the color will be enough to infer what pieces are moved.
  + It is assumed that the king’s initial position will be filled in the interface’s field
* EnPassantMove
  + This class is very similar to BasicMove as it only contains a final position for the pawn
  + However, it would not have been enough to use BasicMove as you need to also remember to remove the taken piece from the board. Denoting this via a separate class makes the implementation easier
* PromotionMove
  + Finally you are allowed to promote a pawn, the most elaborate move you will make.
  + It contains a field file which denotes the file the pawn will wind up in, as well as a promotion choice field.
* HistoryMove
  + This class does not implement the Move interface, it is simply a class that holds a move and a piece name to keep track of the history of moves in order to determine en passant and castling rights.
    1. Pieces

The model also contains an interface Piece with the position, color and piece name. Each class implementing it will have a function getAllValidMoves which returns the list of moves allowed by that piece. The classes that implement Piece are Bishop, King, Knight, Pawn, Queen and Rook. They each have their function returning the valid moves, but written in a way that reuses duplicated code from functions in Piece, since the difference between a Rook and a Queen is small implementation-wise.

* + 1. Board

Moves and pieces are all used in implementing functionalities for the board. A board (or the state of the game) will be determined by the color of the current player making their choice, the set of pieces on the board and the history of moves. It could be argued that it is inefficient to hold the entire history of moves in the state of the board since you only care about en-passant and castling, but it simplifies the implementation of the engine, which is an important goal we’re aiming to accomplish, readable and easily understood code will be preferred over fast but unintelligible code.

The main functions the Board will implement are:

* pieceAt
  + This function receives a position and returns a nullable Piece, which will be null if on that position on the board there is no piece, that piece otherwise.
* move
  + This is another core function, it receives a Move object and implements that move, modifying the state of the Board in accordance with the specification of the move
* getAllValidMoves
  + This function is very important especially in developing the algorithm, it returns all of the possible moves any piece of the current color can make.
* getState
  + This function returns whether the game is over or not. If yes it returns the result of the game as black win, white win or draw.

There are of course, many additional classes and functions in the actual source code of the project, but they are mostly implementation details that could have been approached differently. The presented classes and respective functions are most likely present in any engine you will research, and careful implementation is key in ensuring a problem-free execution.

* 1. **The Algorithm**

There is currently only one class in this package, namely MinMax which as the name suggests is a min max implementation of the proposed engine. Right now there is alpha-beta pruning and iterative deepening implemented, two optimizations that greatly improve the performance of the engine. The code is written in such a way that it is easy to choose whether an optimization is used or not, as comparisons will be desirable at the end of the implementation.

The class implements two key functions, the function evaluate() which returns the estimated score of the state of the board without computing any moves, which is currently a weighted sum of the pieces, positive weights for white and negative weights for black. The following are the weights used, note that the king is given no weight since a game can not exist without either king.

* Pawn: 1
* Bishop: 3
* Knight: 3
* Rook: 5
* Queen: 9

This very simple evaluation function yields good initial results, however it will be a good source of optimization later during development as half the algorithm is just tweaking the evaluation.

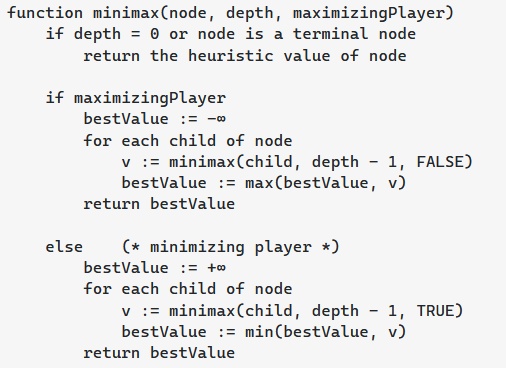


Fig. 4.3.1: Min-Max pseudocode

The second function implemented is getBestMove, which is very likely the most interesting and important function in the entire project. At its core it is a min-max algorithm with a stopping condition (maximum depth). It relies on the observation that, based on the evaluation function we determined earlier, white wants to maximize the final result while black wants to minimize it. This algorithm is closely related to the game tree as it can also be considered a case of dynamic programming on the game tree, albeit still a heuristic since the entire game tree is too large to consider.

* 1. **Running the engine**

I implemented a basic CLI for playing against a human since frontend is currently not the focus of the application. You will at every step be notified what move the CPU has computed and you will receive a list of possible moves, from which you choose the index of the move you want to play and press enter. At the same time you also receive an ascii representation of the board in order to have an easier time deducing what is happening.

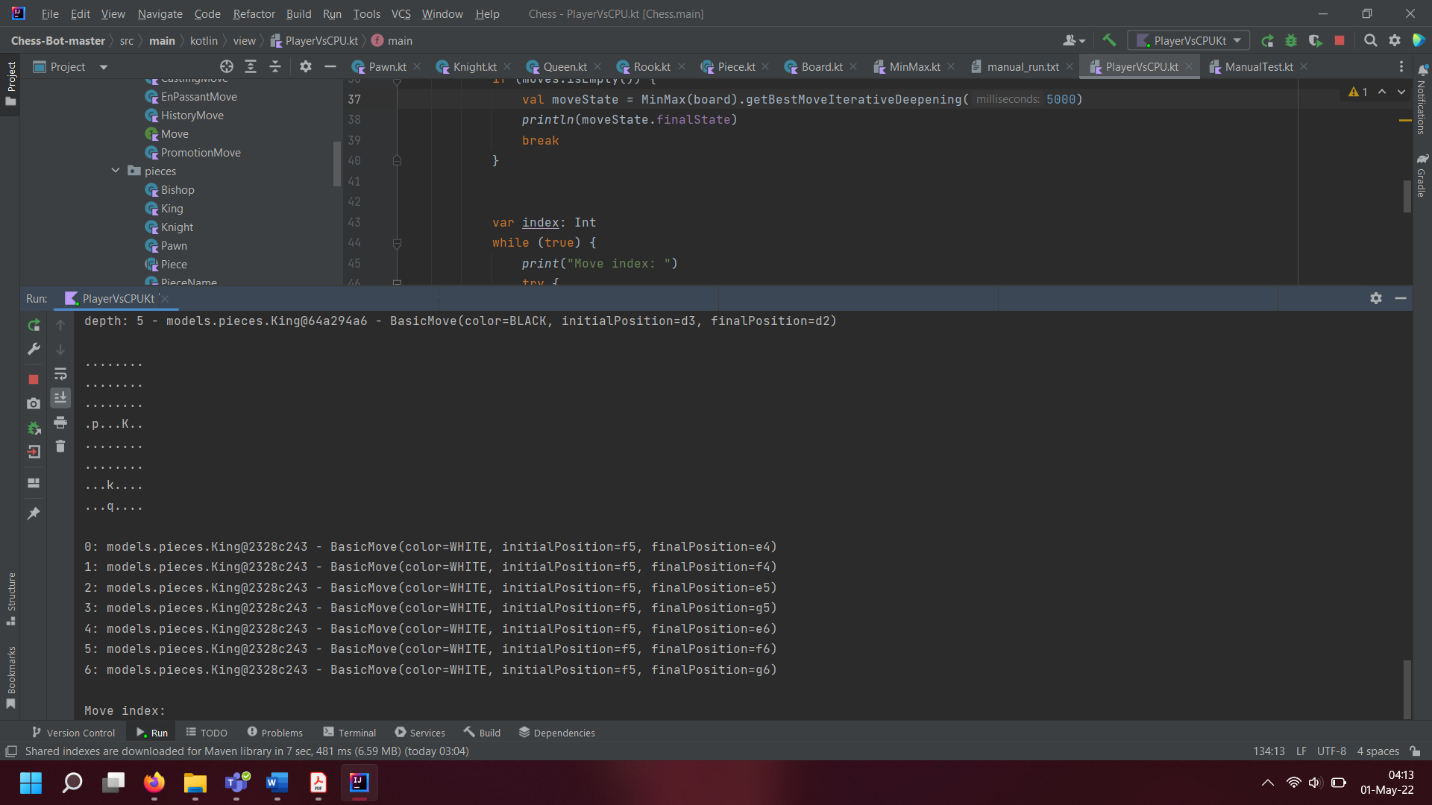


Fig. 4.4.1: The interface of the application

Text is not yet properly handled, it only server as a very simple implementation that can be tested quickly to assess the performance of the algorithm, while I myself assessed the state of the game by replicating it on lichess.org.

1. **Results**

Initial results for the engine are very promising, after only a skeleton implementation of the most basic approaches the chess bot has not lost against me.



Fig. 5.1: Computer Analysis by Stockfish

In the early game the engine struggles to gain any advantage, the random nature of its moves and its sole consideration being the amount of pieces on the board allows the player to steadily gain a positional advantage that can last well into the middlegame. However, I am not very good at chess and I do not spot mistakes that make me lose pieces in only a few turns. Because of this, the curve takes a sharp dive into black territory as I lose several pawns and a minor piece, a loss that persisted for a long time. The algorithm is able to mostly keep its advantage, only for a short time losing it by blundering its knight, however I blunder my own bishop not long after. At the end of the match the engine is able to produce a queen, an advantage that should suffice in beating a single king, but I can continually evade his queen moves since the shallow depth of the game tree considered does not enable the engine to employ the tactic required to mate the king. Had the game continued it would likely have resulted in a draw (by 50 moves without a capture), however it got so far we might as well call it a win. It is really impressive that with such a small implementation it is able to defeat a human player, who knows how easy of a task it will be with a few more tricks under its belt.

**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