BABEŞ-BOLYAI UNIVERSITY CLUJ-NAPOCA
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
SPECIALIZATION COMPUTER SCIENCE

DIPLOMA THESIS

An Exploration into a Chess Minimax Algorithm Implemented in an Object-Oriented Language

Supervisor

lect. Cojocar Dan

Author

Crăciun Ioan Flaviu

2022

UNIVERSITATEA BABEŞ-BOLYAI CLUJ-NAPOCA

FACULTATEA DE MATEMATICĂ ŞI INFORMATICĂ

SPECIALIZAREA INFORMATICĂ

LUCRARE DE LICENŢĂ

Explorarea unui algoritm de şah minimax implementat într-un limbaj de programare orientat pe obiecte

Conducător ştiinţific

lect. Cojocar Dan

Absolvent

Crăciun Ioan Flaviu

2022

Abstract

Chess is a complex and old game that has eluded humanity for hundreds of years, with no access to an oracle that shows us the perfect move at each step. That is until recently, when for the last couple of decades computers have consistently been defeating the world champions. In this thesis I explore the makings of a chess engine and make use of modern fast processors which even with simple implementations yield good results.

The first chapter of this thesis introduces the reader to the field and elaborates on the goals of the thesis. The second chapter explores the state of the art in chess engines, the various types of programs, be they neural network based, or classical backtracking algorithms. The third chapter explains and explores the optimizations implemented, their purpose and motivation on why it would be required. The fourth chapter explains the technologies used, their advantages over alternatives and how they were useful in developing this thesis. The fifth chapter dives into the implementation of the entire project, both frontend and backend. It walks through the most important parts, the model, the algorithm, the overall structure and worthy implementation details. The sixth chapter explores a few games I played against the program, draws conclusions and expands on further work that can be implemented for even better results.

This work is the result of my own activity. I have neither given nor received unauthorized assistance on this work.

Table of Contents

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) [1]. 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 I 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.

Chess engines can work well without perfect heuristics, they can just build massive game trees and perform the minimax algorithm with various improvements such that they may easily be able to challenge human players.

In this thesis I employ several improvements over the basic minimax algorithm and compare their efficacy in developing an AI capable of defeating average humans. The techniques used are alpha-beta pruning, iterative deepening, quiescence search, transposition tables and parallelization.

Human players will be able to play against the engine through an angular-based web application, one which communicates with the engine via the Spring Boot backend that exposes the methods implemented by the engine. The optimizations will be described and detailed in the third chapter, each of which are implemented by the engine. The fourth chapter will introduce the technologies used and their benefits against alternatives, and how they were particularly useful in the development of this thesis. The fifth chapter will describe the practical part of the thesis, the structure of the frontend side and for the backend. The sixth 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.

2. State of the art in chess engines

This chapter describes the current best implementations of the problem I 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

2.1. Elo Rating

The Elo Rating [2] 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 R_A and player B has rating R_B is $E_A = \frac{1}{1+10^{\frac{R_B-R_A}{400}}}$. Similarly, for player B the expected rating is $E_A = \frac{1}{1+10^{\frac{R_A-R_B}{400}}}$.

Supposing player A achieves S_A points, the new rating of player A will be $R'_A = R_A + K * (S_A - E_A)$ where K Is tipically 16 for masters and 32 for weaker players.

2.2. AlphaZero

In 2017 the research company DeepMind unveiled AlphaZero [3], 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.

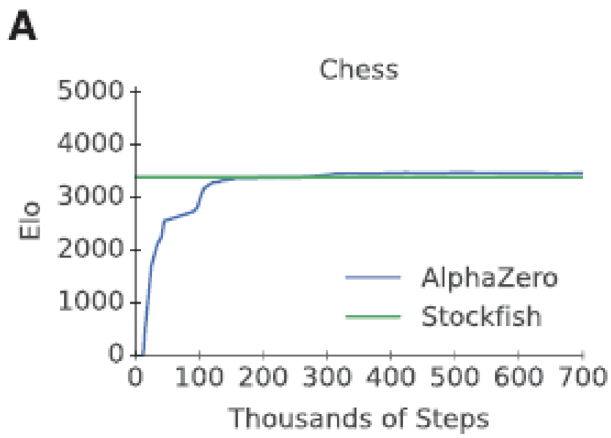

Figure 1 Evolution of Elo rating for AlphaZero [4]

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 cannot 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.

2.3. 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 [5] as the tool that evaluates human moves and gives a very good idea of a player's quality, even among grandmasters.

2.4. 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 [6]. 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 [7], 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.

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 networks take the king crown of chess engines.

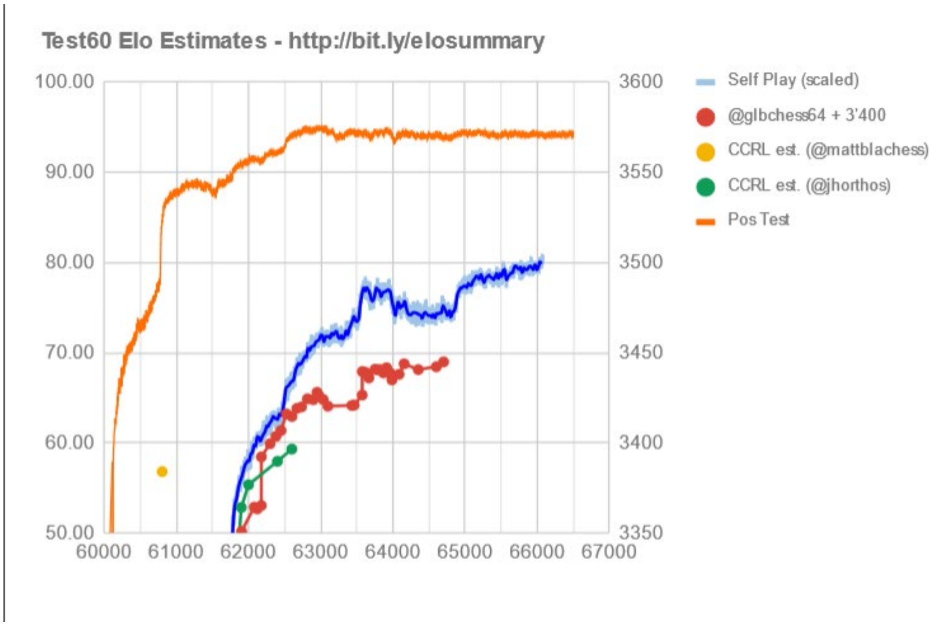

Figure 2 The evolution of Leela's Elo by various metrics [8]

Leela Chess also manages the impressive feat of nearly perfectly solving endgame scenarios [9], further strengthening its viability as a neural network version of a very accurate chess engine.

3. Engine Optimizations

The simplest implementation one can consider for an engine is a fixed depth minimax. While that would indeed be a mathematically correct algorithm, in practice it would be far too inefficient and bad at estimating good moves. In this chapter I explain the optimizations implemented in my algorithm, such optimizations either improve the running time of the algorithm, or the quality of the moves considered.

3.1. Minimax Algorithm

Before we can explore all the tricks we can employ, we must establish the basic idea of our algorithm. Minimax [10] is a backtracking solution based on the observation that given an evaluation function of the state of the board, the white side aims to maximize the score and the black side aims to minimize it. The algorithm is not new, it has been analyzed by the legendary Von Neumann as early as 1928 [11]. In this algorithm a high score means white is likely to win the game, while a low score means black is likely to do so. It is a good starting point for such a program, and the pseudocode of the algorithm can be observed below.

```
function minimax(node, remaining_depth, color)
      if remaining_depth = 0 or node is a terminal node
            return the heuristic value of the node
      if color is white
            bestValue := -infinity
            for each child of node
                  value := minimax(child, remaining_depth-1, black)
                  bestValue := max(bestValue, v)
            return bestValue
      else
            bestValue := infinity
            for each child of node
                  value := minimax(child, remaining_depth-1, black)
                  bestValue := min(bestValue, v)
            return bestValue
```

Figure 3 Minimax pseudocode

Notice the stopping condition is the depth of the game tree, it would be mathematically correct to never stop and compute until every path has been analyzed, but the algorithm would never end within our lifetime. Because of this it is needed to figure out a way to choose a stopping condition, usually a maximum depth in such a way that I get as good results as possible in as little time as possible.

3.2. Alpha-Beta Pruning

The game tree for chess is especially wide. This means that at each step the number of nodes on that depth increases exponentially. Any possibility of decreasing the width of the tree is very promising, and the simplest and most widely applicable is Alpha-Beta [12]. 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. Such a situation can occur when you consider a node that tries to maximize its parent, but its children minimize its value, and its value has been minimized so much that it can no longer improve the parent since a larger value already exists.

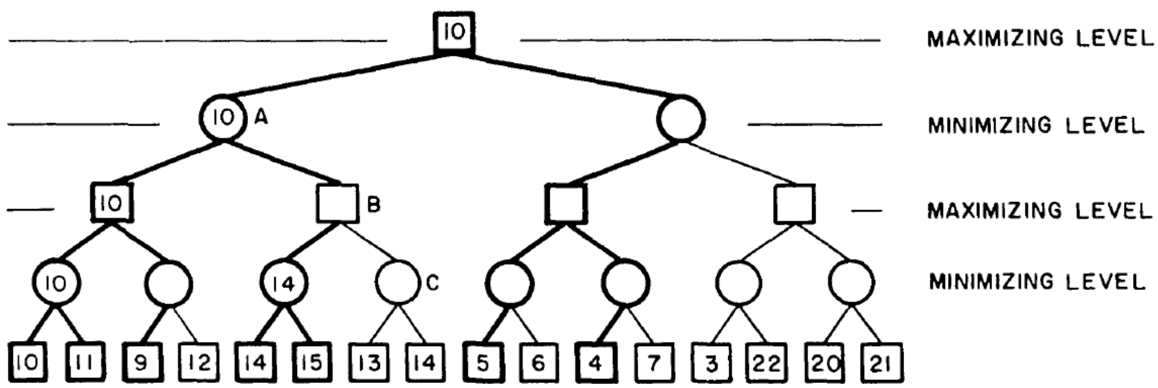

Figure 4 Alpha-Beta Pruning demonstration [13]

Notice the empty nodes do not need to be calculated anymore because, for example, node B is at least 14 which cannot improve the value for node A since node A minimizes itself and already contains a 10.

Implemented naively the technique is quite good, it yields great improvements in execution length. However, if one can arrange the best moves to always come first, pruning will happen earlier and even more time will be saved. Because of this, the better move order heuristics I implement the better alpha-beta pruning will work.

3.3. Iterative deepening

It is hard to know exactly to which depth you want to analyze the game tree in advance. If you go for a fixed value for the duration of the game, you will encounter scenarios where it would've been too quick to finish that depth, and more could've been achieved, or that it would take much too long to finish. For that reason, iterative deepening [14] proposes that I start with a small enough depth, usually 1, and work our way up progressively until I've spent a set amount of time.

This is not as time consuming as it sounds, consider the fact that from each node you will usually have many options to pick from. This results in a game tree that is very wide, and the number of nodes in the last level far surpasses the number of nodes in all the other nodes combined. That is also easy to see in the picture below, even though early depths are checked more than once, exponential growth makes the number of processed nodes in the end almost linear.

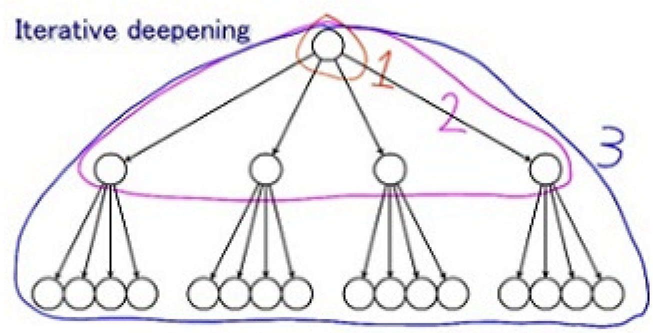

Figure 5 Iterative deepening example [15]

3.4. Quiescence Search

Parsing the game tree naively, that is up to a fixed depth leaves the engine vulnerable to the horizon effect. That is the fallacy that only considering a depth of 4 would not protect against a tactical sequence of moves of length 5. While that could usually be solved by simply considering a longer depth, remember that depth increases the number of nodes considered exponentially and it could be unhelpful to try so.

Another reason quiescence is so important is that when a board is evaluated, the board is assumed to be "quiet". That is no moves that can disrupt the state of affairs heavily can be made, and an evaluation of simply the sum of weights of all pieces can more accurately describe who has the advantage. The problem is, not all board states are quiet. Imagine that on the last level of the game tree we encounter a situation where a queen can be attacked by a pawn, simply adding up their weights will inaccurately report the advantage for the player with the queen.

Quiescence search [16] simply proposes that when we reach the depth limit, we still evaluate the states that are not quiet, up to a reasonable depth. One obvious way to do this is trying out all possible captures, returning the evaluation of the resulting board after a series of captures. This solves the issue outlined above, and although not a perfect approach (what about checks?), it is could be better than not having it.

It turns out that most of the time is spent quiescing, thus pruning the quiescent tree is also valuable, and the implementation is rather straight-forward, similar to the normal minimax. During experiments quiescent nodes made up between 50% and 99% of the nodes processed, and although it may seem like the extra time spent is not worth, it still might improve the quality of moves over the alternative since one depth less is not as bad when you have good evaluation for leaf nodes.

It is important to not forget to include the null move in the quiescent search. This describes the situation where there are captures available, yet the correct move is to not go into any of them [17]. An incorrect implementation will include the standing pat in the evaluation, that is the static evaluation, which in this implementation would be the sum of the values of the pieces on the board.

3.5. Transposition Table

Often in chess there are multiple orderings of moves that lead to the same situation, ultimately leading to evaluating the exact same position more than necessary. The solution to this issue, the transposition table [18], is a technique known from dynamic programming, namely keeping a hash table of a state of a game and associating to it the score of that state. Then when starting the recursion for a certain state I first lookup the table, and if the state exists return its value.

The hash table of a state can be computed in several ways, in this implementation I've chosen the Zobrist hash [19]. This computation associates to each square on the grid, for each possible piece a random bitstring at startup, and then it XORs the associated bitstring for each piece for each position on the board containing a piece.

```
function initialize()
	table := a 64x12 2d array // 12 represents the number of unique pieces
	for i = 1, 64
		for j = 1, 12
			table[i][j] := randomInt()
	black_to_move := randomInt()

function hash(board)
	hash_value := 0
	if board.black_to_move
		hash_value := hash_value XOR black_to_move
	for i from i to 64
		if board[i] is not empty
			j := the piece at board[i] //between 1 and 12
			hash_value := hash_value XOR table[i][j]
	return hash_value
```

Figure 6 Zobrist hash pseudocode

Although the hash of the board is important, another thing to consider is castling rights, en-passant move rights, all which might differ depending on the paths taken through the tree. You could also associate a bitstring to each castling right and xor it with the answer, as well as en-passant positions. That is notably not too important, as usually, especially with shallow trees (small depth), these rights tend to be the same, so it's not very often that this happens.

Transposition tables will speed up the execution time of the algorithm massively, and synergizes well with iterative deepening, as entries from previous runs could be used in the new run. You could also limit the size of the table such that memory will not be filled, and implement various replacement strategies in case the size of the table has been reached.

3.6. Parallelization

Unlike most algorithms we are used to, alpha-beta minimaxes are notably difficult to parallelize. The reason is that in alpha-beta not all children of a node are considered. By running multiple nodes in parallel we risk doing computations for unnecessary subtrees, thus wasting much of the time we would gain by running things in parallel.

I have however attempted an implementation of threads in such a way that not many such subtrees are considered. I pick the first two nodes in the subtree, run them in parallel, update the alpha and beta values after both are finished, then run the rest of the nodes sequentially.

The supposition was that the alpha-beta cutoff will often be nearer the end of the list of children rather than the beginning.

There are ways to implement parallelization even for alpha-beta algorithms, the Young Brothers Wait Concept [20] is one such technique. Running the first nodes until a beta cutoff has been reached sequentially, then parallel execution for the rest of the nodes is the idea, but there are critiques of the results reported, mainly due to inefficiencies of alpha-beta in the author's implementation.

3.7. Heuristics

As mentioned in the alpha-beta subsection heuristics are of high importance, notably move ordering and static board evaluation. Since the first one yields better pruning even simple heuristics can be helpful in reducing runtime. I implemented Most Valuable Victim - Least Valuable Aggressor, that is I prioritize moves that attack pieces that are more valuable than the piece attacked. It was used in MBChess, an engine that made use of move generation at hardware level using this technique [20]. Intuitively attacking a queen with a pawn makes it very likely to be a valuable move to consider. Castling, promotion and en-passant moves are also promoted, while king moves are discouraged (in early-game this is good, in late-game king moves could be valuable but even then, lots of care has to be taken). This helps in pruning quiescence as well since this tactic provides a clear order of attacking moves, which quiescence deals only in these.

For the static board evaluation, I opted for a simple approach, namely adding the weights for each white piece and subtracting the weights for the black pieces. I went with the classic weights used by humans:

- Pawn – 1
- Knight – 3
- Bishop – 3
- Rook – 5
- Queen – 9

Obviously, the king doesn't receive a score, as there will never be a point where not both kings are on the board. This evaluation has the benefit of being extremely fast to compute, however the downside is that it is a bit too simple. Nothing about pawn islands, castling rights, relative position between pieces is considered, although there could be a benefit to doing so. However, the algorithm works well even with this simple approach, as ultimately it is a good enough delimiter for less skilled participants.

4. Technologies used

In this chapter I present the technologies that I used in my project, their purpose and their benefits.

4.1. Kotlin

Kotlin [21] is one of the newest JVM languages recently released, it aims to be an upgrade to the old and clunky Java, it heavily removes the omnipresent Java boilerplate code by making use of an excellent type inference system, and promotes many high-level features which often reduce several lines of code to a single line or a function call. And the impressive feat is that it does all that while remaining similarly fast during runtime (Kotlin performing slightly slower [22]).

As proof of Kotlin's advantages over Java I've written a small piece of code showcasing a class for complex numbers where two numbers are added and displayed to the screen.

```
class Complex {
    4 usages
    Double real, imag;
    3 usages
    Complex(Double real, Double imag) {
        this.real = real;
        this.imag = imag;
    }

    1 usage
    Complex add(Complex c) {
        return new Complex( real: real + c.real,  imag: imag + c.imag);
    }

    @Override
    public String toString() {
        return String.format("%f+%fi", real, imag);
    }
}

public class Main {
    public static void main(String[] args) {
        Complex c1 = new Complex( real: 0.1,  imag: 0.2);
        Complex c2 = new Complex( real: 0.2,  imag: 0.1);
        System.out.println(c1.add(c2));
    }
}
```

Figure 7 Java implementation

Notice the verbosity of the code, one has to write lots of keywords to express something as simple as a constructor which initializes the fields of the class, or a main function that can run the program.

```kotlin
class Complex(val real: Double, val imag: Double) {
    fun add(c: Complex): Complex {
        return Complex( real: real + c.real,  imag: imag + c.imag)
    }

    override fun toString(): String {
        return "$real+${imag}i"
    }
}

fun main() {
    val c1 = Complex( real: 0.1,  imag: 0.2)
    val c2 = Complex( real: 0.2,  imag: 0.1)
    println(c1.add(c2))
}
```

Figure 8 Kotlin Implementation

Notice now Kotlin's version of the same program, there's only 15 lines of code as opposed to 24. They're overall shorter and just as easy to understand as Java.

This comparison serves as a small hint of how Kotlin can be much more efficient in expressing the same ideas as Java, with a more modern design that is not as susceptible to common pitfalls, making it an ideal general-purpose language that can completely replace Java in your programming needs.

The reason I chose this language over a more obvious alternative such as C++ or Python is the ease of developing the model architecture for the chess model, as well as the speed of execution which is relatively high for its capabilities. Most performant engines use C++ because it allows programmers to use object-oriented programming, but speed still close to that of C. I avoided C++ in this thesis because it's a more difficult language to maintain, develop and program overall.

Kotlin is more high level and provides many common patterns which in C++ they have to be implemented by hand. On the other hand, most toy engines use Python due to its ease of use, a dynamically typed language that is generally perfect for most proofs of concept. However, I believe Kotlin is similar in its ease of use, and as an object oriented, statically typed language it's much more robust and reliable than Python. It's also typically much faster, a trait that is highly desired in the pursuit of the most efficient engine.

4.2. Spring Boot

Spring's main important selling point is the way it facilitates inversion of control and dependency injection. Inversion of control [23] implies the inversion of a natural flow of operations in an application. In classical frameworks there's the application that controls the framework (e.g., calls methods from it), but what Spring does is it provides the tools to allow the framework itself to control the application and provide its flow for it. This is most evidently felt in the way that classes are initialized, we find ourselves no longer writing the constructors manually, but we annotate the fields we wish to be automatically populated and the framework will populate them itself at startup. This also provides easy dependency injection as we have control over what objects are being initialized with by their annotation.

Spring Boot [24] is the de facto REST API solution of choice for JVM developers which builds on top of Spring, and it is fully supported in Kotlin as well. It takes an opinionated view on Spring which enables developers to just run the project with minimal fuss (no more deploying WAR files).

If you choose to use the Spring framework, but not Spring Boot, you are required to go through many hoops to deploy your application. Your configurations will be very long and verbose and using Tomcat manually adds difficulty, as it is an old tool with relatively complex and non-user-friendly installation steps. Using Spring Boot it can be as simple as just one annotation and you're ready to write controllers and services and any sort of tools you need in your application, as can be seen below.

```kotlin
package ro.ubb.flaviu

import ...

@SpringBootApplication
class ChessApplication

fun main(args: Array<String>) {
    runApplication<ChessApplication>(*args)
}
```

Figure 9 The skeleton configuration for a Spring Boot app

As the backend of the application will be conceptually quite simple, only a handful of endpoints exposed, Spring Boot is the ideal and obvious choice in avoiding complication and just starting coding the controller that allows interaction with the engine.

4.3. Angular

Angular is one of the most popular Javascript frameworks out there due to its ability to structure components in a hierarchical manner by design, its extensive set of tools that provide easy implementation for common use cases (e.g., routing, http calls) and its ease of use in both small and large scale projects.

I preferred Angular over React because for a chessboard it is natural to use the hierarchy of components, using the ngFor directive we can easily embed 64 squares in a board. I find Angular's structure easier to work with, as React has odd control flow which is harder to master, while something that provides structure by design and has intuitive state management is perfect for the project.

Angular is also based on Typescript [25], an extension (superset) of Javascript which enables developers to write large-scale applications more easily. Typescript builds on top of Javascript a module system, classes, interfaces and a type system. All this leads to a more approachable language for developers used to statically typed languages such as Java or Kotlin, and most of the popular Javascript frameworks support Typescript nowadays.

5. Engine implementation

In this chapter I present the implementation of the application and the technologies used in achieving it.

5.1. Specification

The project will serve as a way for humans to face the chess engine in a complete game. You will be able to play against it and test your skill against the program, either as white or as black. The user interface will be simple, you will simply click on a piece to select it and click on the landing square to move that piece there. After several seconds the bot will play its own move and you will have control again, until draw, win or loss.

The application will be separated between a frontend project, based on angular and a backend project 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 (an old laptop) and the backend can be run on a fast computer.

The engine will implement all of the techniques described in the previous chapter, overall leading to an algorithm able to analyze the position several moves deep, leading to a level of play that is desirably comparable to novice humans.

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 (rooks move orthogonally, bishops diagonally), the possible moves will include the non-trivial:

- Castling (and lack of rights to castle)
- En-passant
- Promoting

These are the moves which require more attention during implementation, as they have complicated preconditions and postconditions. For example, in castling one must take care that no piece exists between the king and rook, the king or any piece between the king and rook is not under attack and the king and rook have never moves up to now. The application will correctly assess the state of the game (win, loss, draw), return all possible moves for the player or engine to choose from and allow two parties (CPU or human) to carry out a game.

The desired outcome is a backend that makes efficient use of a computer's resources, while returning good moves in a short amount of time.

5.2. Model

The model of the application is a central part of the implementation, as chess has many moves that do not adhere to a simple pattern, moves which complicate the OOP approach and make it employ various design patterns to suit exceptions. 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.

5.2.1. Moves

Moves are organized as classes that implement a simple interface. The base Move interface contains an initial position of the piece moved, the final position of that piece, the color of the piece moved, a Boolean flag indicating whether a capture has been made, and a score estimating its usefulness (capturing a queen with a pawn is usually better than a king move). Below are all the types of moves one may make during the game of chess.

- 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.
  - It simply implements the fields in the base Move class as they are enough to describe this type of move
- 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, as well as a high score since castling is very useful.
    - It is assumed that the king's position will be used in the interface's position fields
- EnPassantMove
    - This class is to BasicMove, with the isCapture field always being true, and the score is a fixed high value (en passant moves are desirable)
    - 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.
    - It also holds a reference to the attacked piece if it exists in order to successfully "un-move" a move during backtracking.

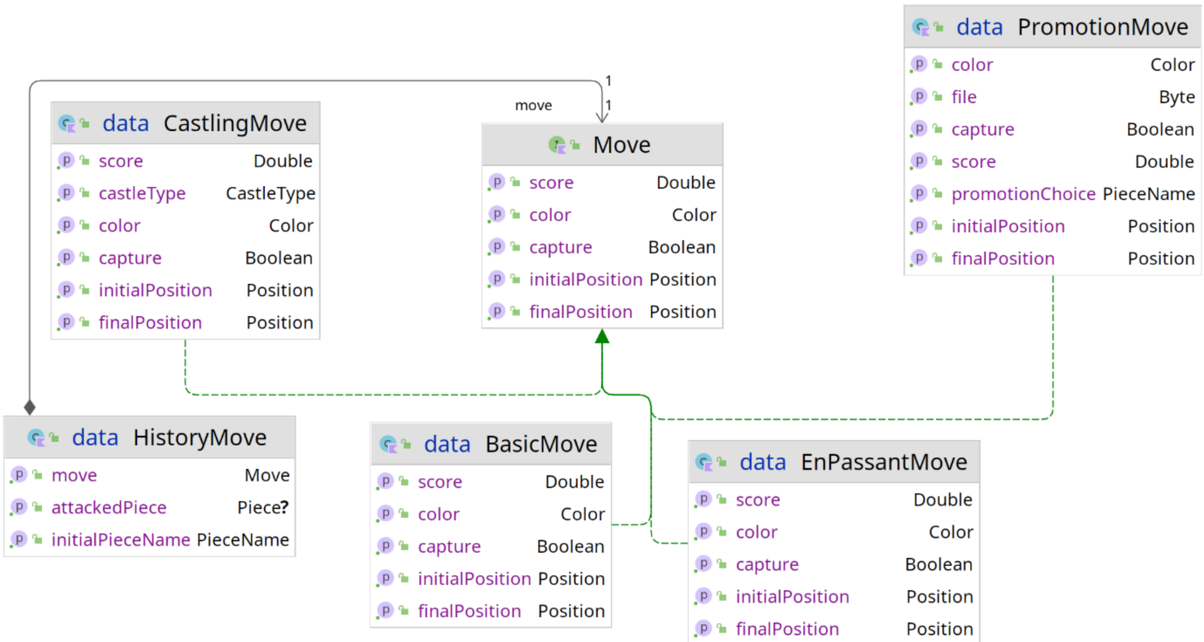

Figure 10 The class diagram for the moves

Notice the class diagram denotes the inheritance of the 4 classes from the Move interface, as well as a HistoryMove which contains one move. All properties are also neatly laid down and easy to observe, thus compiling a useful representation of possible moves in chess.

5.2.2. Pieces

The model also contains an interface Piece with the position, color, piece name and score (a value tied to each piece). 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.

The scores of each piece are as mentioned in subsection 3.7, which are not only used in estimating the static evaluation of a board, but also used for determining the order of moves to consider (remember alpha-beta pruning benefits from good heuristics). The order of moves considered follows the most valuable victim, least valuable aggressor heuristic also mentioned in

3.7, by associating the score of the move with the difference between the value of the victim and the value of the aggressor times 10.

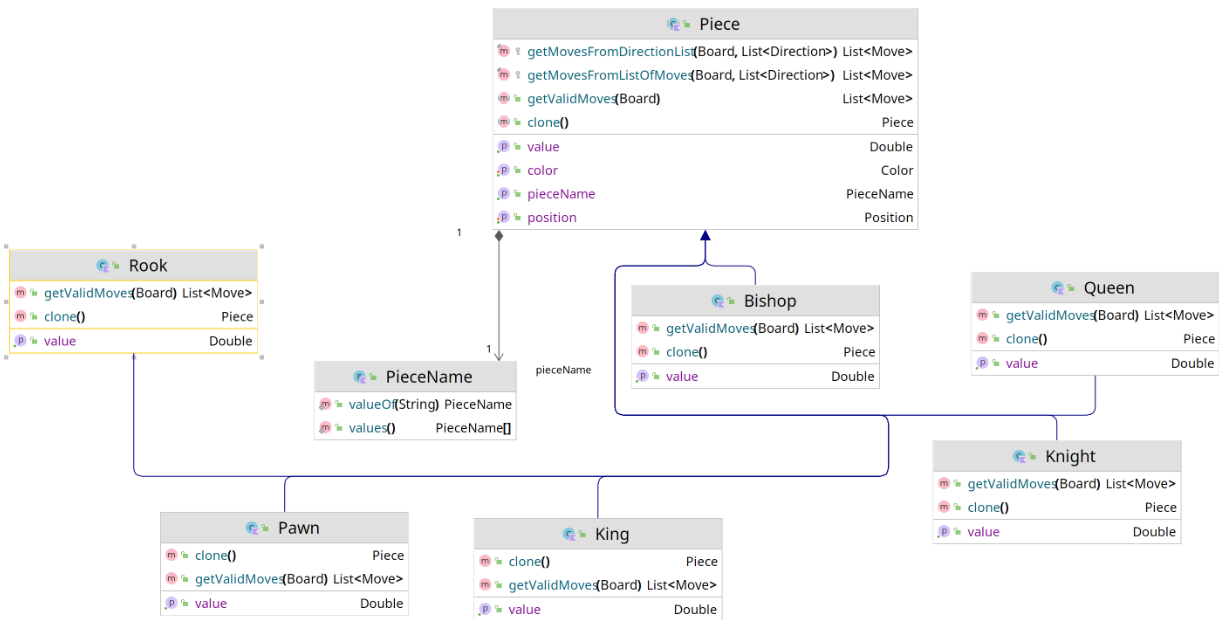

Figure 11 Pieces class diagram

The diagram for piece classes also shows the functions and properties implemented, notice there are two helper protected functions in Piece that generate moves for the other pieces by reusing much of the code you would normally duplicate in Rook and Queen for example.

5.2.3. 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. The history of moves is used in determining castling rights, en-passant opportunity and maintaining the stack of moves with extra info such that it will be easier to perform the back-track operation in backtracking.

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
- unmove
    - This function takes the move on top of the history of moves and un-implements it, reverting the state of the board to that before it
- 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.

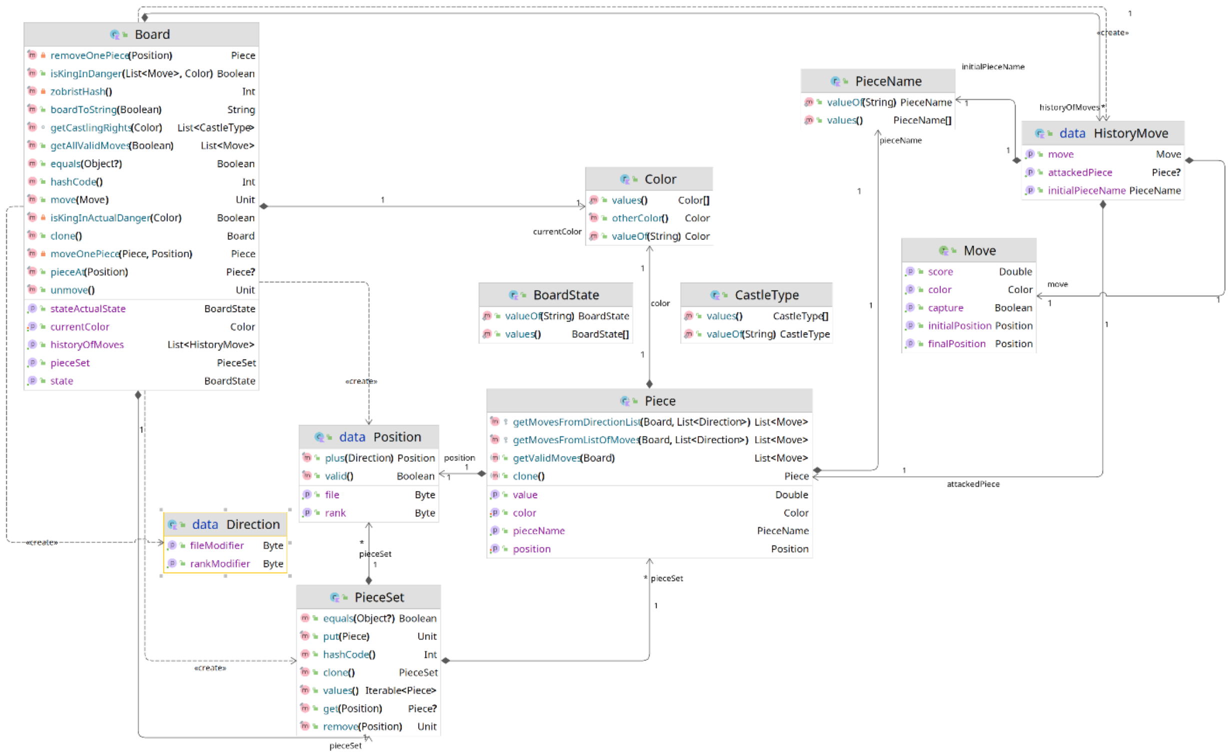

Figure 12 Board class diagram

In this class diagram we may notice the interaction between pieces and the board, while move is only used as an external variable and is not present anywhere in the properties of the classes. There are several functions in Board, but as mentioned they either help with code reuse, or implement a helper functionality such as clone or Zobrist hash.

5.3. The Algorithm

There is only one class in this package, namely Minimax which as the name suggests is a min max implementation of the proposed engine. As mentioned all of the optimizations enumerated in chapter 3 are implemented, with varying degrees of success. 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.

This very simple evaluation function yields good results, and although it may be a good source of optimization, I chose to go for a simple, fast, easy to understand metric by which to estimate who is more likely to win. Of course, more information like pawn structure, piece synergy, protection of the king, and so on could be helpful, but it ultimately boils down to very chess-specific information, and the purpose of this thesis is to explore general techniques in a chess-context, not chess in particular.

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 a limited game tree, building it indirectly through recursion.

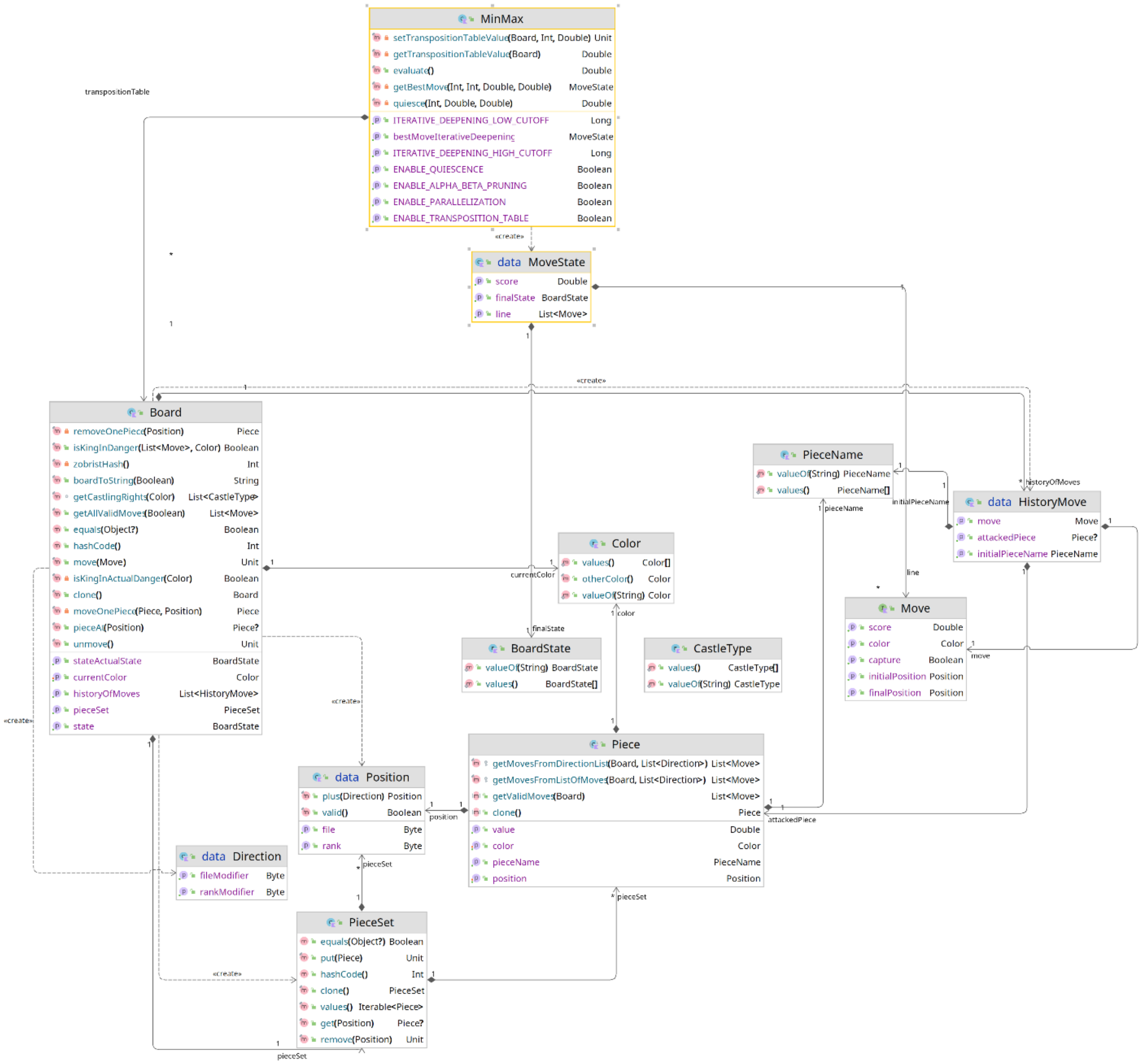

Figure 13 Minimax algorithm class diagram

We may notice from the class diagram the relationships between the MinMax class, the Board, the Move class, as well as other helper classes. The MinMax class creates MoveStates which contain Moves, it also contains many Boards through the addition of the transposition table that maps boards to numbers, as well as the current board being analyzed.

The following are details related to the implementation of each technique listed in chapter 3, including any configurable parameters related to them.

5.3.1. Alpha-Beta Pruning

Alpha-Beta pruning was a straight-forward implementation of the pseudocode from subsection 3.2. Alpha-Beta was notably also used in the quiescence algorithm since that one dominates the runtime. There are no configurable parameters for alpha-beta.

5.3.2. Quiescence

Quiescence was implemented in exactly the same manner as the normal minimax, except it filters all the moves that are not captures so that only captures are considered. The maximum depth is the variable parameter, is could be unlimited, or limited up to a small number. Going for a small number yields bad returns from quiescence and was found to promote bad moves, but going for a very large number (effectively unlimited) also yielded bad moves since the depth of the actual minimax algorithm was stumped by the very large number of quiescent nodes. A sweet spot in practice was found to be ~10.

5.3.3. Transposition Table

The implementation for transposition table in this algorithm was also quite simple, yet very effective in reducing runtime (an observed ~ x2 decrease in the total time spent). I simply opted to cache each state encountered using the Zobrist hash, limiting the size of the map to 1 million, and checking whether the state exists in the table at the beginning of the recursion. Extra care had to be taken such that before each run in the iterative deepening procedure the hash table is reset, else states evaluated with depth=1 would yield unusable results at larger depths.

The configurable parameters for the table are the size of the table and the maximum depth up to which I update the table, yet being lenient with both variables was found to be optimal (that is large values).

5.3.4. Parallelization

As mentioned earlier, I did not achieve satisfactory results by implementing parallelization in this algorithm. There was far too high of an overhead starting so many threads, such that whatever gain there was from multiple processors was lost. Also, extra subtrees were often computed that were not necessary, overall, often increasing the total time spent. There are however two configurable parameters, the number of threads spawned by each node and the depth cutoff after which no threads would be spawned.

5.4. Server

Since the application was split into a backend and a frontend project, there of course needs to be a way to expose the methods such that a chess application can be interacted with. The server was written in Spring Boot, a very simple and well-known tool that can be used in the Kotlin programming language. The following are the methods exposed by the server:

- newGame
    - This method returns to the client an identifier for a game with the board in the starting formation.
    - It is called at the beginning of a game to initiate a game in the backend's memory such that the state of the game doesn't reset upon refresh.
- getGame
    - This method returns the board of a game given an identifier.
    - It called upon first page load when an url with a game identifier is loaded.
- computeMove
    - This method receives a game identifier and starts the minimax algorithm to compute the best move.
    - It returns the new board with the move played on it after being persisted in the backend's memory.
- move
    - This method submits a move and a game identifier and returns the board with the move applied to it.
    - It removes code duplication so that the frontend does not implement the same feature, and makes use of the heavily tested backend code.
- getMoves
    - This method returns all the valid moves one may make given a game identifier.
    - The client will call this method and display to the user the moves they are allowed to make based on the returned list.

This rather small set of operations is just enough to be able to play against the algorithm and test any moves. No other features from Spring Boot are used as this focus has been set on algorithm performance rather than user experience, yet the user experience is smooth nonetheless.

5.5. Frontend

A simple client written in Angular has also been provided to easily interact with the engine. The user is first landed on a page where they can choose whether to play white or black, as well as the configuration of the engine running. Following that they will be displayed the chessboard and two boxes containing the missing pieces for white and for black. In order for the user to make a move they have to click on a piece, and then click on a piece with a green circle in the middle representing the destination (or from king to rook in case of castling). The square on the initial position and final position of the moved piece will be displayed with different background to make it simple for the user to realize what was the last move. There is also a service class that makes http calls to the backend which were outlined in subsection 4.4.

5.5.1. Models

As data has to be sent from the backend to the frontend and vice-vera, some specification on what the objects contain is required. The following are the data-transfer objects (DTOs) implemented both in the backend and in the frontend:

- Board
    - This is the obvious one, it is the board that contains the piece list, the current color (whether white or black is to move), the history of moves and the state of the board (unfinished, draw, white win or black win).
- ExecuteMove
    - This class is used to submit the required information to the Move operation in the backend, namely the board, the actual move and in case of pawn promotion, the choice of piece promotion.
- HistoryMove

- This class is implemented to simply be used inside Board and to send back to the backend the same object as received.
- Move
  - Moves are interacted with a lot between calls, as such the same fields a move contains in the backend are present here as well: initial position, final position, color and the class name (to translate from base interface to specialized move such as CastlingMove)
- Piece
  - The piece contains just like in the backend, a position, color and a class name which helps specialize the class that winds up processed by the algorithm.
- Position
  - The position is also used as a component of other DTOs, and only contains the rank and file.
- Options
  - The parameters used during the engine's board state evaluation. They include flags for enabling/disabling alpha-beta pruning, quiescence searching, the transposition table and parallelization, as well as two numbers which tune how long the algorithm analyses a position.

5.5.2. Components

This subsection presents the 6 components of the application, how they are conceptually implemented and used.

5.5.2.1. Chessboard

This is the main component of the application which contains inside it the entire functionality. It is a div that arranges its subitems in an 8x8 grid, and indeed it contains 64 subcomponents which represent each individual square. It implements clicking functionality such that by clicking on a piece all possible destinations are highlighted, and choosing one of them will move the piece to that position. It then communicates with the backend to process the move and cue the engine to begin its own move computation and return the new board.

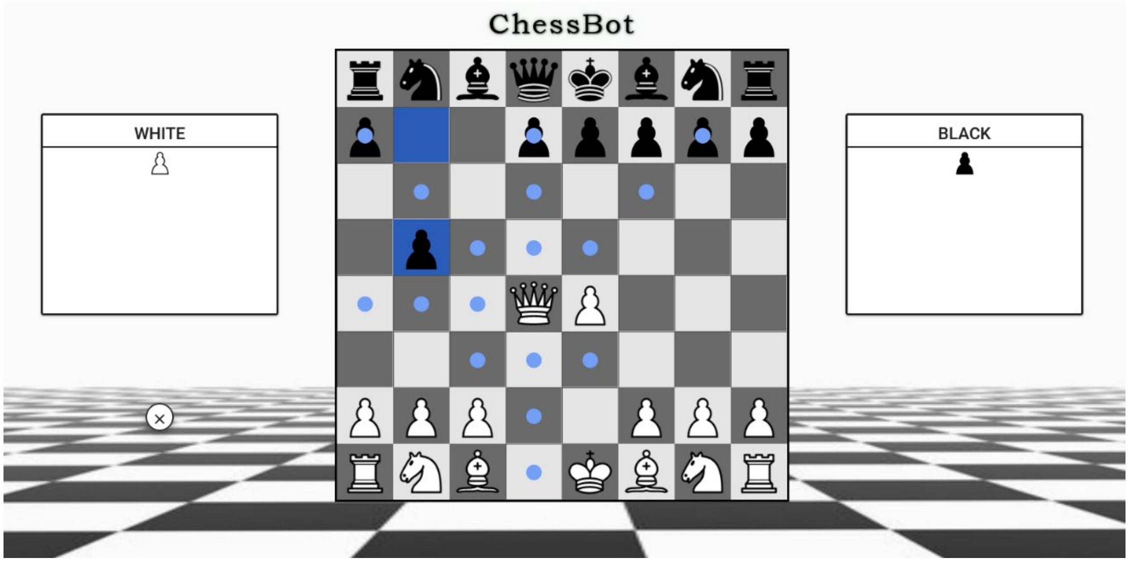

Figure 14 The chessboard component in the game page

5.5.2.2. Chess Piece

This component is an individual square within the 8x8 squares in the grid, it contains a colored background (light or dark square), possibly a piece, and possibly a green dot signaling a potential move. It might also present a blue background in case this square has been involved in the last move performed on the table. The component receives a position, piece and a flag as input, the flag marking whether a piece may move there or not. It then simply parses these inputs to produce the correct looking square for the user.

5.5.2.3. Promotion Choice

This component is used to allow the player to choose what piece they want to promote their pawn into. It is opened by the Chessboard component as a dialog and clicking on a piece will return the choice back to the chessboard component and then complete the info necessary for the move.

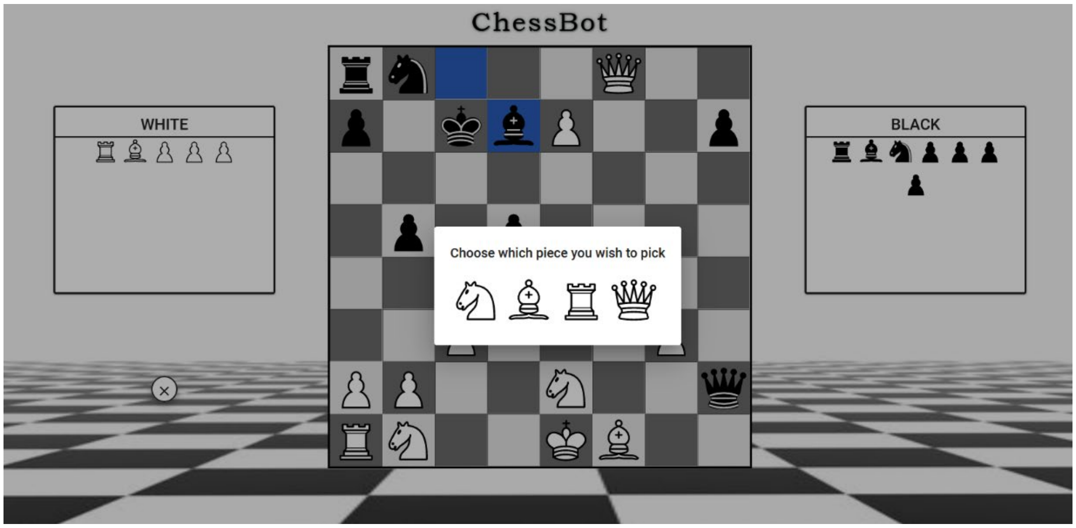

Figure 15 The pawn promotion

5.5.2.4. Game

During a game this component contains the chessboard as well as a button to go back, and a list of pieces that each color is missing. It accesses a game identifier stored in local storage which the backend uses to keep track of the state of the game. It is also the page which the previous two figures have been showcasing.

If not in a game and the user has just entered the application, they are greeted with a dialog through which they configure the necessary data to play the game of chess.

5.5.2.5. Choose Color to Play

This component is what's displayed in the first dialog that prompts the user to choose their color. It only contains the two buttons for the color and allows the user to setup the bot parameters before starting the game.

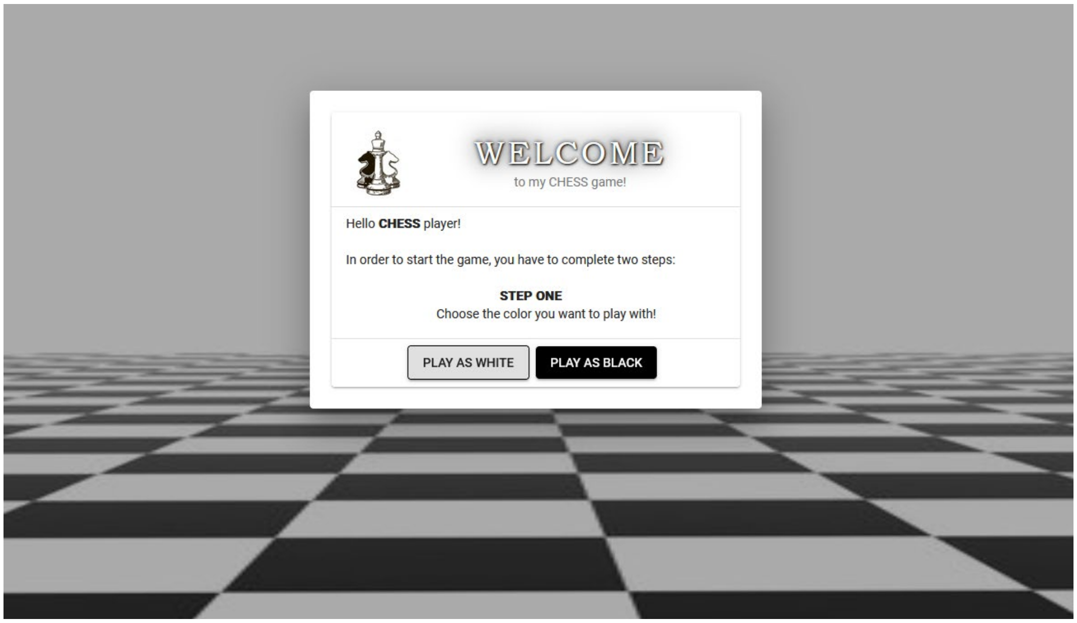

Figure 16 The color dialog

5.5.2.6. Setup

This component allows the user to choose which optimizations to run the bot with. The defaults are empirically found to be the best performing ones, but one can try any variation they prefer and test against human input (i.e., play against the bot).

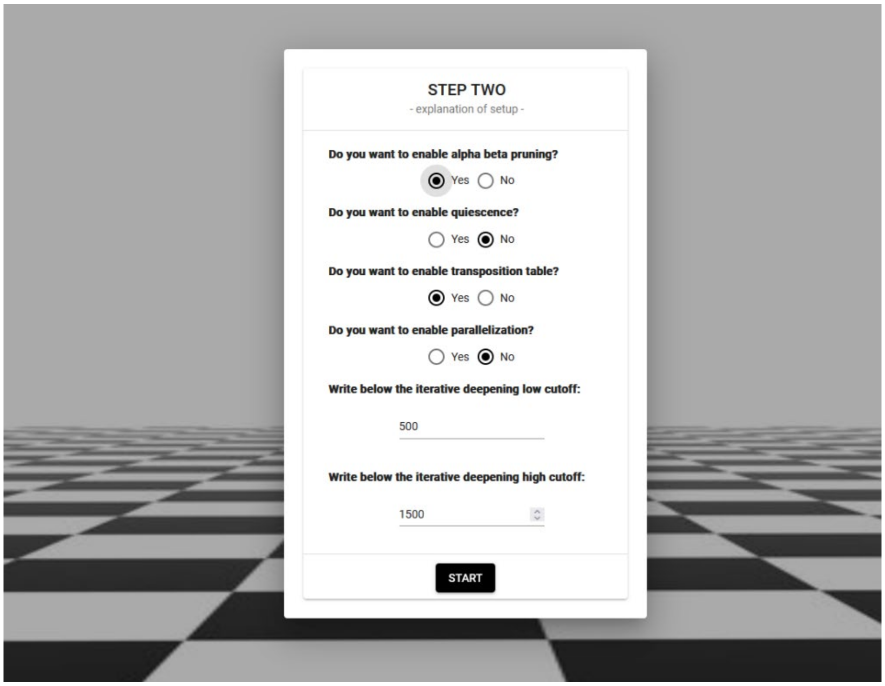

Figure 17 The setup component

6. Results and Conclusions

6.1. Algorithm Complexity

Chess is a particularly difficult game to analyze. To drive in the point of that here's a few interesting facts about the game:

- Shannon estimated that there are 10^{120} valid different games of chess in existence [26]
- After each player has made a move 2 times there are 197,281 different games, and after each has made their third move there are 119,060,324 such games.
- Computing a perfect strategy for n×n chess requires time exponential in n [27]

From this we can understand why engines usually only analyze the position up to the depth of around 5-20 (it may vary depending on the position, time allotted and pruning efficiency). Consider the minimax algorithm which has a complexity of $O(b^d)$. This means the numbers of moves grows approximately proportional to b (the average breadth of the game tree) raised to the power of d (the number of moves computed). It is improved by the alpha-beta optimization, which at worst case it performs just like the basic minimax algorithm, on its best case it is $O(\sqrt{b^d})$ because the first player's moves must be studied to find the best one, but for each, only the second player's best move is needed to refute all but the first (and best) first player mov. Alpha-beta ensures no other second player moves need be considered. On average, Alpha-Beta yields $O(((b-1+\sqrt{1+14b+b^2})/4)^d)$ [28] when nodes are considered in a random order, and the game tree is uniform with binary leaf-values. This is of course, not the case of chess, but one can definitely observe the efficiency of alpha-beta as opposed to simple minimax.

Quiescence search's complexity is $O(a^b \times c^d)$ where a is the average breadth of the search tree, d is the depth up to which the search is performed, c is the average breadth of the quiescence seach and finally d is the depth up to which the quiescence is performed. One immediately notices that if c and d are significantly large the number of actual games analyzed is much smaller, thus this optimization's benefit has to be considerable for improvement to occur over the natural algorithm.

Finally, parallelization complexity is difficult to compute, since my implementation on a mathematical level negates alpha-beta and reduces the basic minimax complexity to O($\frac{b^d}{c}$) where c is the number of cores. Since c is usually small on home computers, this is not a useful optimization.

6.2. Optimizations Performance

In this subchapter the performance-wise impact of each optimization will be studied. The starting position will be considered, and the total amount of time to reach the highest depth will be reported, after which comparisons may be drawn. The engine will run on a Windows 11 machine with a 6 core Ryzen 5600H CPU. Since randomness will affect the runtime of the algorithm there will be 5 attempts with each configuration each and the smallest time window will be reported. A limit of 15 seconds will be imposed, meaning no more searches will be performed if more than 15 seconds passed.

- Simple Minimax

A baseline is required to more accurately compare the results. This will be the simple algorithm with no optimizations enabled. It turns out that it takes the engine 14.92 seconds to reach depth 5 (5 moves analyzed).

- Alpha-Beta Pruning

A significant improvement is achieved by alpha-beta pruning, only 3.74 seconds to depth 5 are required in the best case which is around 4 times faster than simple minimax.

- Transposition Table

Transposition tables yield an impressive improvement as well, the best number is 6.60 seconds to depth 5. That is more than half of the 14.92 seconds that no optimizations will give, which is a very good result for a simple implementation. Furthermore, combining alpha-beta with transposition boosts the best time to an even lower 2.33 seconds. This optimization is impressive since in early game there’s more different paths that lead to different games than in endgame, consider that late in the game there’s fewer moves available which make the ratio of different states to paths much smaller.

- Parallelization

The implementation of parallelization in this engine does not offer a significant improvement, but tinkering with parameters yielded 13.45 seconds. It seems that there is no great advantage even when not using alpha-beta pruning, which is difficult to pinpoint the reason of, but could be any of windows not being efficient in JVM thread spawning, the number of nodes in the subtrees varying wildly or large overhead in thread creation (there's many variables being copied between threads).

Surprisingly, when combining alpha-beta pruning with parallelization the best time is reduced to 2.29 seconds, which is less than just either of the two. Combining alpha-beta, transposition and parallelization leads to similar timing, 2.37 seconds.

- Quiescence

This optimization kills depth, in 15 seconds only 3 moves being analyzed (which finishes in 0.17 seconds). That may be detrimental as a depth of 3 even with good quiescence is not enough to confidently offer good suggestions by the engine. On first glance however the quality of play hadn't incurred a loss, and didn't make too obvious blunders. While this method of analysis would suggest quiescence is an awful disadvantage imposed on the engine, the level of play is still high.

6.2.1. Against Stockfish

The following games were played against Fairy-Stockfish on lichess.org. Each level has been tackled until the algorithm has incurred a loss, in an attempt to quickly analyze the performance. Only alpha-beta pruning and the transposition table optimizations were used (the ones found to be an obvious performance increase), as well as a low cutoff for iterative deepening of 500 milliseconds and a high cutoff of 2500 milliseconds. A low cutoff means that if the algorithm finished a depth and has been running overall for at least 500ms then it will finish, while a high cutoff means that if the algorithm is still running and 2500ms have passed then it will forcefully finish.

The following subchapters make use of the PGN notation. In the 19th century, the world chess federation (Fédération Internationale des Échecs, known as FIDE) popularized and imposed the use of Algebraic Notation (AN) to record moves made in chess games. The standard was devised in order to be easy to interpret and read by humans and be shared universally. Given the advent of computerized chess, Portable Game Notation (PGN), essentially a wrapper around standard AN, was devised as a plain-text computer-processible format for recording games. [29]

6.2.2. Stockfish Level 1

PGN: 1. e4 h5 2. h3 e5 3. Nf3 Qf6 4. Nc3 g5 5. d3 Bh6 6. Nd5 Qc6 7. d4 Nf6 8. Nxf6+ Qxf6 9. Nxe5 Bf8 10. c3 c5 11. Be3 a5 12. Qb3 Rh7 13. Rd1 Qg7 14. a3 f6 15. Qd5 cxd4 16. Nxd7 Nxd7 17. Qxd4 Bc5 18. Qd5 Bxe3 19. Bc4 Ra7 20. e5 Rh8 21. Qe6+ Kf8 22. O-O Bb6 23. Bb3 fxe5 24. Kh2 Ra6 25. a4 Bc5 26. Qd5 g4 27. Kg1 Rd6 28. Qxc5 Nxc5 29. hxg4 Rxd1 30. gxh5 Rxf1+ 31. Kh2 Rxh5# 0-1

In this game the stockfish algorithm has managed to build a positional advantage, but the engine developed quickly capitalized on material mistakes and the end was quite abrupt from the moment the white queen has been lost.

6.2.3. Stockfish Level 2

PGN 1. d4 c6 2. c4 Qb6 3. b3 f5 4. e3 Nf6 5. Ba3 e6 6. Qd2 h5 7. h3 Nh7 8. c5 Qc7 9. Nf3 d6 10. cxd6 Bxd6 11. Ne5 Bxe5 12. Qd1 Bxd4 13. Qxh5+ Kd7 14. exd4 g6 15. Qxg6 Kd8 16. Nd2 b5 17. Qh5 Qb7 18. Rc1 Rg8 19. g3 Nf8 20. Bd6 Ng6 21. Bb4 Qc7 22. h4 Ne5 23. Bxb5 Ng4 24. Ba4 e5 25. O-O exd4 26. Rce1 Ne5 27. Rc1 d3 28. Rfe1 Ba6 29. Qh6 Ng4 30. Qe6 Rh8 31. Nc4 Bxc4 32. Qxc4 Qd7 33. Re6 Qg7 34. f3 Ne5 35. Rxe5 Qxg3+ 36. Kh1 Rxh4+ 37. Qxh4+ Qxh4+ 38. Kg2 Qxb4 39. Kf1 d2 40. Rd1 Qc3 41. Rxd2+ Qxd2 42. Re6 c5 43. Rf6 Qc1+ 44. Ke2 Qb2+ 45. Ke3 Qd4+ 46. Ke2 Qxf6 47. Kd1 a5 48. Bb5 Qa1+ 49. Kc2 Qxa2+ 50. Kd1 Qxb3+ 51. Ke2 Qxb5+ 52. Kd2 Qa4 53. Kd3 Qb3+ 54. Kd2 f4 55. Kc1 Ke8 56. Kd2 Qxf3 57. Ke1 Qh1+ 58. Kf2 Qc1 59. Kg2 Qc2+ 60. Kf3 Qh2 61. Kg4 Nd7 62. Kg5 Qf2 63. Kg4 Rd8 64. Kh3 Qg3# 0-1

In a similar manner to the previous game, Stockfish managed to grab a positional advantage early on as well as a small material advantage, at a point even being 4 moves away from checkmate. It blundered its advantage away soon after that and the engine has managed to slowly capture white's pieces until white remained with only its king which got mated eventually.

6.2.4. Stockfish Level 3

PGN: 1. d4 e6 2. c4 Qf6 3. Nc3 a5 4. h4 Na6 5. Nf3 Bb4 6. Bd2 Rb8 7. g4 h6 8. g5 Qf5 9. a3 Bxc3 10. Bxc3 Qg6 11. h5 Qh7 12. e3 Qf5 13. Ne5 hxg5 14. c5 Qe4 15. Rh3 b6 16. Bb5 Qg2 17. Bf1 Qa8 18. cxb6 cxb6 19. Be2 Rb7 20. Nc4 Qb8 21. d5 Rh7 22. dxe6 d5 23. Qc2 Rh6 24. Ne5 fxe6 25. Rf3 Nc5 26. b4 Nd7 27. Nc6 Qh2 28. Kd2 d4 29. Bb2 dxe3+ 30. Kxe3 g4 31. Nxa5 Rc7 32. Nc6 gxf3 33. Bf1 g6 34. Rd1 Qxh5 35. Rd6 Qg4 36. Bc4 Qg5+ 37. Kd3 Ne5+ 38. Bxe5 Rxc6 39. Qa4 Qf5+ 40. Kd2 Qe4 41. Rxc6 Bd7 42. Rxe6+ Ne7 43. Qc2 Qh4 44. Ke1 Bxe6 45. Bd3 g5 46. Qc7 Kf7 47. Qd8 Bb3 48. Qc7 Re6 49. Bf5 Qh1+ 50. Kd2 Qd1+ 51. Ke3 Rxe5+ 52. Qxe5 Qe2+ 53. Kd4 Qd1+ 54. Ke3 Qe2+ 55. Kd4 Qxe5+ 56. Kxe5 b5 57. Bg4 Bd1 58. Ke4 Ng6 59. Ke3 Nh4 60. Kd2 Ba4 61. Ke3 Bd1 62. Bd7 Ng6 63. Bxb5 Nh4 64. Kd2 Be2 65. a4 Ng2 66. Bd3 Nh4 67. a5 Ng6 68. Bxe2 fxe2 69. a6 Kf8 70. a7 Kg8 71. a8=Q+ Kg7 72. Qa1+ Kg8 73. Qd4 Ne7 74. Qc4+ Kf8 75. Qxe2 Nc8 76. Qe4 Ne7 77. Kd3 Ng8 78. Qf5+ Ke8 79. Qxg5 Kf8 80. f3 Kf7 81. b5 Nf6 82. f4 Ke7 83. b6 Ke6 84. Qb5 Nd5 85. Qc6+ Kf7 86. Qxd5+ Kg7 87. b7 Kg6 88. Qe4+ Kh5 89. b8=Q Kg4 90. Ke3 Kh4 91. Kf3 Kh5 92. Qh8# 1-0

The lengthy third game proved the tipping point for the engine implemented in this paper. In the first half it seemed like the course of the game would look like the other two, with an early positional disadvantage, while the engine would gain material which might prove enough to snatch the win. However, Stockfish made a comeback and in turn captured all of black's pieces, mating him on the 92nd move. While the engine lost to Stockfish level 3, it wouldn't be far off to call it of a similar level considering it held off impressively until the 66th move where a blunder took away black's advantage, where we could very easily have seen it conclude like the second game.

6.3. Conclusions

Unfortunately, two techniques implemented did not yield great results. Quiescence search led to inaccurate moves and shallow game trees due to the predominance of the search space, and parallelism, while improving the bare algorithm and the algorithm running alpha-beta pruning, it could not contribute to the best performing setup, at least on the initial board. Alpha-beta pruning has managed to reduce runtime by a factor of around 4, which is expected as most moves are awful and not considering most of them is very beneficial. The transposition table also yielded runtime benefits of a factor of more than 2, highlighting yet again how often positions repeat in chess. The heuristics which improve move ordering also played a great part in bringing alpha-beta to its score.

In my opinion this engine is capable of defeating novice players, it can defend itself against immediate threats and is even fun to play against and try to outsmart. Such a program written in a high-level object-oriented programming language definitely presented both benefits and disadvantages. The main benefit was ease of implementation due to OOP abstractions which allowed easy code reusability, clear hierarchies and design patterns and reliability in using pre-existing data structures such as maps and arbitrarily sized arrays. I believe writing a chess engine in Kotlin as a learning experience is a great opportunity and one may have the chance to develop skills they otherwise wouldn't, as applying such a powerful language to a non-trivial algorithm will highlight the language's strengths. In spite of the language's high-level features overhead I would say the endeavor has been fruitful, as a viable novice competitor was the result of a few techniques implemented to a simple minimax algorithm.

6.4. Further Work

The chess algorithm implemented in this thesis yielded promising results, yet further work would undoubtably improve the quality of play tremendously. More research on more efficient parallelization would finally break the inefficiency of only using one CPU to solve the task, and depending on the device's number of CPU cores the runtime would be sped up several times over.

The heuristics employed were also quite lackluster, and indeed I've only done a minimal implementation of things like static evaluation or move ordering. More aggressive pruning, better evaluation that considers positional as well as material advantage, opening and endgame tables would all improve the results a great deal.

As we’ve seen in the state of the art of chess engines, finding out a method to employ more modern artificial intelligence approach such as neural networks could also be a path towards maximum efficiency. Another idea would be increasing the number of configurable parameters and based on them run evolutionary algorithms that pit the various game engines against each other to find the best parameters you can set.

Overall, much research could help out the results of the program, and although not ideal, the initial results are not bad either. Given more time the program would start climbing the ELO ladder and defeat master players just as easily as Stockfish or Leela Chess.

References

[1] Campbell, M., Hoane Jr, A. J., & Hsu, F. H, "Deep blue," *Arificial intelligence,* pp. 57-83, 2002.

[2] Regan, K. W., & Haworth, G. M., "Intrinsic chess ratings," *Twenty-fifth aaai conference on artificial intelligence,* 2011.

[3] Dong, H., Dong, H., Ding, Z., Zhang, S., & Chang, Deep Reinforcement Learning, Singapore: Springer, 2020.

[4] Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., ... & Hassabis, D, "A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play," *Science,* pp. 1140-1144, 2018.

[5] 29 August 2022. [Online]. Available: lichess.org.

[6] Haque, R., Wei, T. H., & Müller, M. Haque, "Evaluating Leela Chess Zero Against Endgame Tablebases," *Advances in Computer Games,* 2022.

[7] Haworth, G., & Hernandez, N, "TCEC Cup 2," *ICGA Journal,* pp. 100-107, 2019.

[8] "Full Elo Graph," 19 August 2022. [Online]. Available: https://training.lczero.org/.

[9] Haque, R., Wei, T. H., & Müller, M, "On the Road to Perfection? Evaluating Leela Chess Zero Against Endgame Tablebases," *Advances in Computer Games,* 2022.

[10] Dem’yanov, V. F., & Malozemov, V. N., Introduction to minimax, Courier Corporation, 1990.

[11] Von Neumann, J., & Morgenstern, O., Theory of games and economic behavior, 1947.

[12] Knuth, D. E., & Moore, R. W., "An analysis of alpha-beta pruning," *Artificial intelligence,* pp. 293-326, 1975.

[13] Pearl, J., "The solution for the branching factor of the alpha-beta pruning algorithm and its optimality", in *Communications of the ACM*, 1982.

[14] Korf, R. E., "Depth-first iterative-deepening: An optimal admissible tree search," *Artificial intelligence,* pp. 97-109, 1985.

[15] "Iterative Deepening," 19 August 2022. [Online]. Available: https://www.chessprogramming.org/Iterative_Deepening.

[16] Harris, L. The heuristic search and the game of chess. a study of quiescence, sacrifices, and plan oriented play. In *Computer chess compendium.* pp. 136-142, Springer, New York, NY, 1988.

[17] Beal, D. F, "A generalised quiescence search algorithm," *Artificial Intelligence,* pp. 85-98, 1990.

[18] Slate, D. J, "A chess program that uses its transposition table to learn from experience," *ICGA Journal,* pp. 59-71, 1987.

[19] Zobrist, A. L., "A New Hashing Method with Application for Game Playing," *ICCA Journal,* 1970.

[20] Schaeffer, J., "Distributed game-tree searching," *Journal of parallel and distributed computing,* pp. 90-114, 1989.

[21] Samuel, S., & Bocutiu, S., Programming kotlin, Packt Publishing Ltd, 2017.

[22] Gakis, S., & Everlönn, N., "Java and Kotlin, a performance comparison," *Bachelor Thesis,* 2020.

[23] Sobernig, S., & Zdun, U., "Inversion-of-control layer," in *15th European Conference on Pattern Languages of Programs*, 2010.

[24] "Spring Boot," 20 August 2022. [Online]. Available: https://spring.io/projects/spring-boot.

[25] Bierman, G., Abadi, M., & Torgersen, M., "Understanding typescript," in *European Conference on Object-Oriented Programming*, Berlin, 2014.

[26] Shannon, C. E., "A chess-playing machine," *Scientific American,* pp. 48-51, 1950.

[27] Fraenkel, A. S., & Lichtenstein, D., "Computing a perfect strategy for n× n chess requires time exponential in n," in *International Colloquium on Automata, Languages, and Programming*, Berlin, 1981.

[28] Saks, M., & Wigderson, A., "Probabilistic Boolean Decision Trees and the Complexity of Evaluating Game Trees," in *27th Annual Symposium on Foundations of Computer Science*, 1986.

[29] Shah, N., & Prashanth, G., "PGN/AN Verification for Legal Chess Gameplay," 2015.

[30] Boulé, M., & Zilic, Z., *An FPGA Move Generator for the Game of Chess,* McGill University, 2002.