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. dr. 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. dr. Cojocar Dan

Absolvent

Crăciun Ioan Flaviu

2022

Abstract

It isn't often that a centuries old game keeps the entire humanity interested in it such that a large community still discusses and analyzes its depth even today. Yet chess is one such rare example. We used to apply logic and depend on the top players of the world to get accurate assessments of positions, but something changed in the last few decades. The processing power of computers has increased to much that it's been a long time since the best human players are incapable of defeating the top chess engines.

In this thesis I explore the basics in chess engine development and implement my own algorithm which makes use of several well-known techniques that improve performance. The first chapter presents the field and the goals of the thesis. In the second chapter the best chess engines are analyzed, their type which is neural network or minimax. The third chapter presents the techniques used in the project and what they accomplish. The fourth chapter presents the technologies which the project made use of, their benefits over alternatives and their importance in the success of the project. The fifth chapter explains how the project was structured and implemented, the client side as well as the server side. The sixth chapter analyses the efficiency impact of the techniques used, the complexity analysis of each and several games played against different levels of a top-level chess engine.

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 a game that has been played for an estimated 7 centuries, and analyzed and discussed for that many still. Players have gotten better and better with time, yet have been sidelined recently by computers in the capacity of evaluating positions. It was 1997 that DeepBlue defeated the reigning world champion Garry Kasparov [1], and the best engine has only gotten better since. In spite of the complexity of the game, it is ultimately a game between two players that involves zero hidden information and is completely deterministic (meaning no randomness is involved). This makes it relatively accessible for an engine to brute-force its way through many positions in order to determine the best move, as well as implementing heuristics and tricks that improve accuracy. Usually, the top performing chess engines are written in a fast low-level language, C++ mainly, but there is also the option of going the object-oriented route. This thesis will explore the structure and results of writing a chess engine in an object-oriented language, and the language of choice in this project is Kotlin.

Because of how fast modern processors are it is not required to come up with perfect heuristics, one can write a minimax algorithm that builds a large game tree and gradually improve it in order to defeat human players of low to medium skill. Thus, this thesis will implement several such improvements and compare their efficiency, their accuracy and level of play. The following optimizations/techniques are employed:

- Alpha-Beta Pruning
- Iterative Deepening
- Quiescence Search
- The Transposition Table
- Parallelization

The engine will be made available to play against through a web application with the frontend written in Angular, and the backend in Spring Boot which facilitates the frontend's job of displaying the board and performing moves. The techniques will be presented in the third chapter of this thesis, the fourth will introduce the technologies used and their benefits against

alternatives, and how they were particularly useful in the development of this thesis. In the fifth chapter the implementation of the frontend and backend will be detailed which will serve as the practical chapter of this thesis. The sixth chapter takes a look at the implemented chess engine and quantifies its performance and accuracy during play. Lastly, the conclusions are presented and the results interpreted.

2. State of the art in chess engines

In this chapter the best and most popular chess engines are presented in order to introduce the reader to the state of the art. While there's many to choose from, I have settled on the 3 that I believe to be the most important and relevant to current day research.

2.1. Elo Rating

The Elo Rating [2] system was introduced in the 20th century to determine the relative performance of zero-sum games. The formulas it introduces can predict the probability of win /loss for each player. For example, a player 100 points above their opponent in rating should score 64% of points, but a difference of 200 points raises the expectation to 76%.

Because it is a logarithmic scale, 100 points difference between two played presents the same outcome regardless of the absolute score of the two players, whether it is 1300/1400, or 2200/2300. The formula for the expected score of player A with rating R_A and player B with rating R_B is $E_A = \frac{1}{1+10^{\frac{R_B - R_A}{400}}}$. Symmetrically, for player B the expected score is $E_A = \frac{1}{1+10^{\frac{R_A - R_B}{400}}}$.

If A achieves S_A points, that player will gain the new rating of $R'_A = R_A + K * (S_A - E_A)$ with K tipically 16 for masters and 32 for less skilled players.

2.2. AlphaZero

It was in 2017 that AlphaZero was unveiled to the world by Deepmind [3], a neural network algorithm with unmatched performance that with only 4 hours of training it was about as good as Stockfish, and with 9 hours of training it snatched 28 wins out of 100 games against Stockfish, the remainder being draws.

What is remarkable is that it's the first engine based on neural networks that broke the barrier and showed top level skill. AlphaZero has zero knowledge of chess other than the basic rules, yet simply by playing against itself it can outsmart the best engine to date.

Not only can the engine play chess, but it also mastered Go and Shogi and can be adapted to play Atari and board games, an unexpected result that such a general idea is enough to bring record breaking feats in chess.

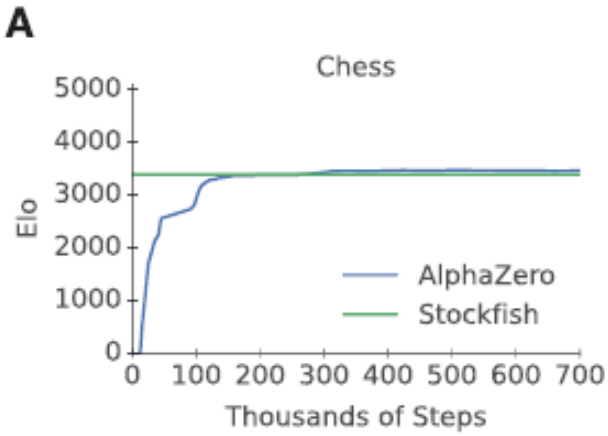

Figure 1 Evolution of Elo rating for AlphaZero [4]

Notice in figure 1 that this engine matches Stockfish early during training and reaches a point where it tops the Elo rating by 100. Unfortunately, the results cannot be used to gauge the Elo rating of the engine in the context of other chess engines as the comparison has only been done with Stockfish, but it certainly shows that it shouldn't be challenged by any other engine given the wide margin.

2.3. Stockfish

The standard algorithm that other engines compare to, Stockfish is arguably the strongest classical engine, using a backtracking algorithm that works best on the CPU (AlphaZero in comparison relies on a strong GPU for training). It has been improved and calibrated heavily over the years and was considered the gold standard in chess algorithms, that is until AlphaZero became the first to defeat it. It is free and open source, which means anybody can contribute to its code base.

Learning from its defeat, in 2020 a neural network version was developed and promises a strong improvement, but its significance is still considered that of a strong minimax engine.

The long-time reigning engine has been the victor in most TCEC (Top Chess Engine Championship) competitions, but it seems that a new engine by the name of Leela Chess is gaining ground and snatching wins from Stockfish.

Even though it's been shown that other better engines exist, Stockfish is still used as the de facto evaluation tool for providing humans with insight in evaluating positions, particularly in Lichess [5], most notably due to its open-source design and age in its field which made it a trusted choice.

2.4. Leela Chess Zero

While AlphaZero made rounds with its results out of the blue, it is unfortunately a closed-source solution developed by a private company, and such cannot be used by the public at large. In order to emulate its success and prove the papers surrounding it, an open-source implementation emerged, by the name of Leela Chess [6]. It played chess by itself, similar to how AlphaZero did, and by the 500th million game it seemed to match Stockfish. While Leela Chess doesn't use a supercomputer like those that Deepmind uses, it runs a distributed implementation that relies on the community to play games locally and contribute to the performance of the engine.

Only a few months after being released, in April 2018 Leela Chess made the first appearance at TCEC as a neural network engine, but sadly performed poorly, only winning one game (due to a crash of the opponent), drawing two and losing the remainder of 25. It improved soon after, and by February 2019 it won the first cup, not losing any game during the entire championship.

As an open-source neural network engine, Leela Chess is the most exciting addition to the field, and many hope that it will be the reigning champ that overthrows Stockfish, and given time it may certainly prove to be that.

Leela Chess presents a better Elo rating estimate than its Deepming counterpart, and on its official website a graph (that can be observed in figure 2) is posted that claim to place the engine at the mark of 3500 points. It is better than Stockfish, though it might still improve, so in time it may prove the dominance of neural network 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

A fixed depth minimax is the most straightforward engine implementation that can considered. Though this method would be accurate mathematically, it would be very inefficient and poor at predicting good moves. In this chapter, I go over the optimizations I employed in my engine, which either speed up the program's execution or improve the quality of the moves it reports.

3.1. Minimax Algorithm

We must first establish the basic principle of our algorithm before we can examine any method we might use. Based on the observation that, given an evaluation function of the state of the board, the white side seeks to maximize the score and the black side seeks to minimize it, minimax [10] is a backtracking solution with alternating computations depending on depth. The algorithm is not new; the renowned Von Neumann studied it as early as 1928 [11]. In minimax a high score denotes that white is more likely to succeed than black, and a low score denotes the opposite. It is a reasonable starting point for developing the engine, and the algorithm's pseudocode can be viewed 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

The depth of the game tree is used as a stopping condition. Although it would be mathematically valid to continue computing until every path is examined, this would mean that the algorithm never finishes. In order to achieve the best outcomes in the shortest amount of time, a stopping condition is necessary, which is typically a limit on the game tree depth.

3.2. Alpha-Beta Pruning

The chess game tree is particularly large. This indicates that the number of nodes at each depth level grows exponentially. Any opportunity to reduce the tree is helpful, and Alpha-Beta [12] is the simplest and most often used option. The method of alpha-beta pruning involves not taking into account nodes in the game tree that, regardless of their value, can no longer change the value of the root node. Such a scenario can arise when a node tries to maximize its parent but its children reduce its value to the point where it can no longer increase the parent's value because a bigger value already exists.

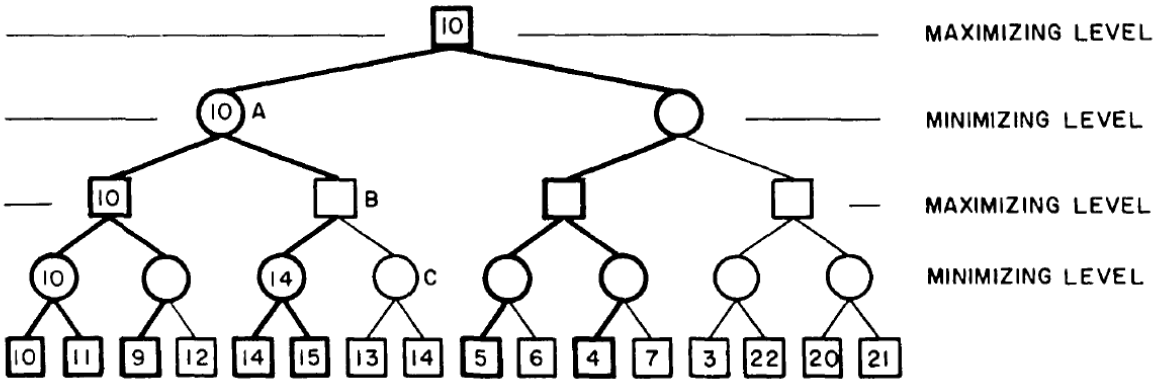

Figure 4 Alpha-Beta Pruning demonstration [13]

For instance, node B is at least 14 and cannot increase the value of node A because node A minimizes itself and already includes a 10; so, the empty nodes do not need to be calculated any longer.

When applied crudely, the approach produces significant reductions in execution time. However, pruning will occur earlier and even more time will be saved if one can set up the best moves to always appear first. Because of this, alpha-beta pruning will function better the better move order heuristics are used.

3.3. Iterative deepening

It can be challenging to determine in advance the precise depth to which you want to investigate the game tree. If you choose a fixed value for the game's depth, you may run into situations where more could have been accomplished in a reasonable amount of time, or where it would take far too long to complete. Because of this, iterative deepening [14] suggests

beginning at a low enough depth, often 1, and gradually increase it until a predetermined amount of time has passed.

Consider that there are typically several possibilities available from each node, so it is not as time-consuming as it sounds. As a result, the game tree becomes extremely broad, with the number of nodes at the final level vastly exceeding those in all other nodes put together. It is also clear from the image below that, despite the fact that early depths are checked more than once, exponential growth causes the final number of processed nodes to be practically linear.

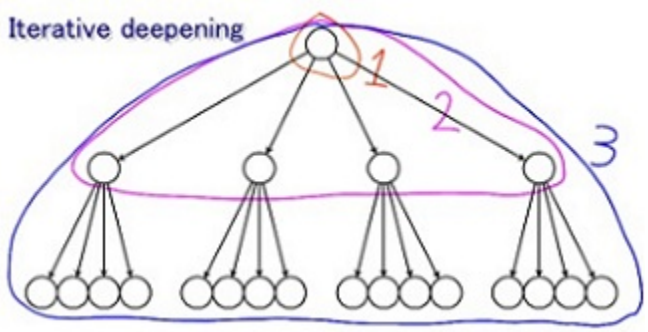

Figure 5 Iterative deepening example [15]

3.4. Quiescence Search

The engine is susceptible to the horizon effect when the game tree is naively parsed up to a fixed depth. That is the fallacy that a tactical sequence of movements with a length of 5 would not be protected against by an algorithm that analyses up to a depth of 4 moves. While in theory this can be resolved by just taking a deeper look, keep in mind that doing so exponentially increases the number of nodes that are factored in.

Because a board is presumed to be "silent" when being examined, quiescence is necessary for this purpose as well. If there are no actions that can significantly alter the current situation, it is more correct to assess who is in a better position by adding the weights of all the individual pieces. Not all board states are quiet, which is the issue. Consider a scenario where a queen can be captured by a pawn on the last level of the game tree; if the weights of the two pieces were simply added, the player who has the queen would not truly have an advantage.

The simple suggestion made by quiescence search [16] is to continue evaluating the states that are not quiet up to a second fixed depth after reaching the depth limit. Trying out every

potential capture and then reporting the evaluation of the resulting board after a series of captures is a straightforward approach to execute this. This resolves the aforementioned problem, and while it's not a perfect solution (what about checks?), it could be preferable to doing nothing.

The majority of the time is surprisingly spent quiescing, therefore pruning the quiescence tree is also beneficial. The implementation is rather simple and is similar to the standard minimax. Between 50% and 99% of the nodes processed during testing were quiescent, and although it may appear that the extra time was not worthwhile, one depth level less is not as problematic when you have good evaluation for leaf nodes, therefore it still might increase the quality of moves over the alternative.

The null move must be considered when performing the quiescence search. This condition is when there are opportunities for captures, but the best course of action is to avoid taking any of them [17]. A proper implementation will incorporate the static evaluation, also known as the standing pat, which in our algorithm would be the total value of the pieces on the board.

3.5. Transposition Table

In chess, there are commonly several possible move orders that result in the same state, which forces players to evaluate the same position more than is necessary. The transposition table [18], a dynamic programming technique used to solve this problem, keeps a hash table of a game's state and associates the score of that state with it. Consequently, I look up the table before beginning the recursion for a specific state, and if the state is there, I return its value.

There are numerous ways to generate a state's hash table, but for this implementation I've opted for the Zobrist hash [19]. At the beginning of the computation, a random bitstring is assigned to each square on the grid for each piece. Then, for each position on the board that contains a piece, the corresponding bitstring for that piece on that position is XORed.

```
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

In addition to the board's hash, you need also think about castling rights and en-passant move rights because these may vary based on the routes you follow through the tree. You can assign a bitstring to each castling right, en-passant positions, then xor them with to the final hash. It's noteworthy that this is not significant because these rights typically coexist, particularly with shallow trees (small depth), and this rarely occurs.

Transposition tables significantly reduce the algorithm's execution time and work well with iterative deepening since they allow entries from earlier runs to be reused in the current iteration. Additionally, you might set a table size restriction to prevent memory from being filled and implement different replacement techniques in the event that the table's size limit has been reached.

3.6. Parallelization

Alpha-beta minimaxes are noticeably more challenging to parallelize than the majority of the algorithms we are familiar with. The issue is that not all offspring of a node are taken into account in alpha-beta. We run the danger of wasting a large portion of the time we would save by executing things in parallel by operating numerous nodes simultaneously.

However, I've tried to come up with a threaded solution that takes fewer subtrees into account. I choose the first two nodes in the subtree, execute them simultaneously, update the

alpha and beta values after they're both done, and then execute the remaining nodes one at a time. It was assumed that the alpha-beta cutoff would frequently be closer to the end of the list of children nodes than the beginning.

Even for alpha-beta algorithms, suitable parallelization can be implemented; one such method is the Young Brothers Wait Concept [20]. The aim is to run the initial nodes sequentially until a beta cutoff has been achieved, then run the remaining nodes in parallel. However, there have been criticisms of the reported findings, primarily because of inefficiencies in the author's implementation of alpha-beta.

3.7. Heuristics

Heuristics, particularly move ordering and static board evaluation, are very important, as was noted in the alpha-beta subsection. Move ordering can be useful in lowering runtime because it results in better pruning. I used the strategy of Most Valuable Victim - Least Valuable Aggressor, prioritizing attacks on pieces that are more valuable than the attacking piece. It was employed in MBChess, an engine that utilized this method of move generation at the hardware level [20]. It would seem logical that using a pawn to attack a queen would be a sensible choice to consider. Similarly valued are castling, promotion, and en-passant moves, whereas king moves are avoided (that improves early game accuracy, and while late game could use king moves, it's still dangerous to walk him around). Due to the tactic's clear ordering of offensive moves (quiescence deals solely in these) it aids in pruning quiescence as well.

I used a straightforward strategy for the static board evaluation, combining the weights of each white piece and deducting the weights of the black pieces. I chose the traditional weights used by most players:

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

Since both kings must always be on the board, it is obvious that the king is not given a score. This evaluation has the advantage of being very quick to compute, but it has the drawback of being a little too simplistic. Although there may be a benefit, nothing concerning pawn islands, casting rights, or relative position between pieces is taken into account by my implementation. Even with this straightforward strategy, the algorithm functions well because it serves as a reasonable evaluation as there's comparatively few states with the same number of pieces on board during analysis.

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.

```
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 give people a chance to play a full game against the chess engine. You will be able to play against it and compete skill-wise with the computer, playing either the white or the black side. The user interface will be straightforward; all you have to do to move a piece is click on it to pick it and then click on the landing square. The bot will play its own move after a few seconds, at which point you will regain control until a draw, win, or defeat.

An angular-based frontend project and a backend project written in Kotlin using Spring Boot will be used to split the application. This prevents duplication of computation-related code, also allowing for the development of potentially numerous clients, such as an Android application. Additionally, while the backend can run on a quick computer, the frontend code can run on a slower device (such as an outdated laptop).

All of the strategies from the previous chapter will be implemented by the engine, creating an algorithm that can assess a situation several moves deep and, as a result, play at a level that is ideally at least comparable to that of inexperienced humans.

The user interface will be straightforward because computing moves is the main goal, but it will nevertheless allow users to move pieces on the board in line with chess rules. The available moves will include the non-trivial moves (castling, en-passant, promotions), as well as the well-known ones (bishops move diagonally, rooks move orthogonally).

Due to their complex preconditions and postconditions, the complex moves require greater care during implementation. For instance, when castling, one must ensure that no piece between the king and the rook exists, that neither the king nor any piece between the king and the rook is under assault, and that the king and the rook have not yet made any moves. The program will accurately determine the game's outcome (win, loss, or draw), return all viable moves for the player or engine to select from, and enable two players (a CPU or a human) to play the game.

A backend with effective resource management that can quickly and accurately return good movements is the intended result.

5.2. Model

Since many of the moves in chess do not follow a clear pattern, the OOP method is complicated and one must use a variety of design patterns to account for exceptions. This places the application model at the center of the implementation. The majority of movements include moving piece A to position P and maybe getting rid of piece B. However, you can castle, which requires moving two pieces at once (queenside or kingside). You can also advance a pawn that requires clarification regarding the position to which it should advance. The history of moves is also significant because it affects castling rights and en-passant availability.

Nevertheless, it has been done, and OOP has demonstrated success in streamlining development, lowering code size, and enhancing readability.

5.2.1. Moves

Moves are categorized as classes that implement a straightforward interface. The basic Move interface shows the piece's initial location, final position, color, a flag that indicates whether a capture was made, and a score showing how useful the move is (capturing a queen with a pawn is usually better than a king move). Here is a list of every possible move type in the game of chess:

- BasicMove
    - The move you will use the most frequently is one that moves a piece from position A to position B. Rook from a2 to c2, or f3 pawn captures piece on g4, are two examples.
    - It merely implements the base Move class's fields, as they adequately represent this form of move.
- CastlingMove
    - The only information contained in this move is the castling kind (queenside vs. kingside), which, along with the color, will be sufficient to determine what

pieces are moved. Castling also results in a high score because it is very valuable.

- EnPassantMove
    - This class is akin to BasicMove except that the score has a fixed high value and the isCapture field is always true (en passant moves are desirable).
    - Nevertheless, using BasicMove alone wouldn't have been sufficient because you also need to remember to take the taken piece off the board. Making this explicit via a different class simplifies implementation.
- PromotionMove
    - As a last move, you are permitted to promote a pawn, which is the most complex one you will use.
    - It includes a promotion choice field and a field file that indicates the file the piece will end up in.
- HistoryMove
    - This class is merely a class that stores a move and a piece name in order to maintain track of the history of moves and establish en-passant and castling rights. It does not implement the Move interface.
    - If the attacked piece is present, it also keeps a reference to it in order to properly "un-move" a move while retracing.

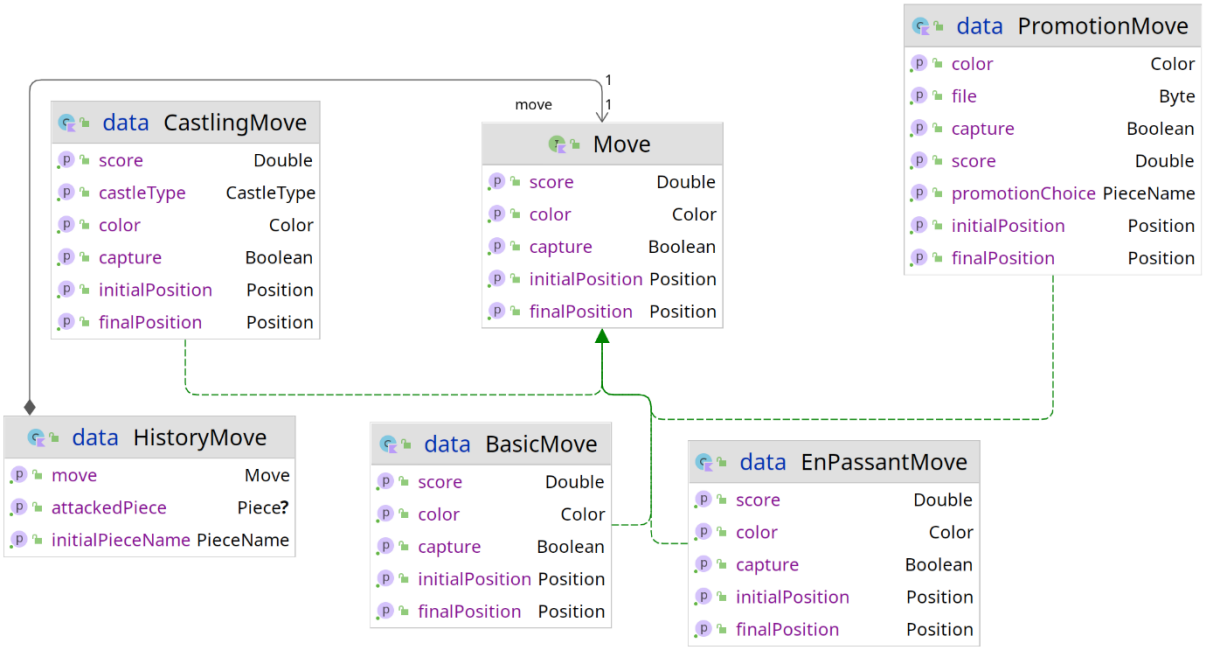

Figure 10 The class diagram for the moves

Take note that the class diagram shows how the four classes that inherit from the Move interface, as well as a HistoryMove that includes one move. Additionally, all characteristics are properly organized and simple to see, creating a meaningful representation of chess move options.

5.2.2. Pieces

Additionally, the model has an interface Piece with the piece's position, color, name, and score (a value tied to each piece). The function getAllValidMoves, which provides a list of the moves permitted by that piece, is present in each class that implements it. The classes Bishop, King, Knight, Pawn, Queen, and Rook implement Piece. Since the implementational difference between a Rook and a Queen is negligible, they have each created a function that returns the valid movements, but the code was reused from functions in Piece.

The scores of each piece are those described in subsection 3.7, and they are used to calculate both the static evaluation of a board and the order of moves to be taken into consideration (remember alpha-beta pruning benefits from good heuristics). By assigning the score of the move to the difference between the value of the victim and the value of the

aggressor multiplied by 10, the order of moves examined follows the most valued victim, least valuable aggressor heuristic previously indicated in 3.7.

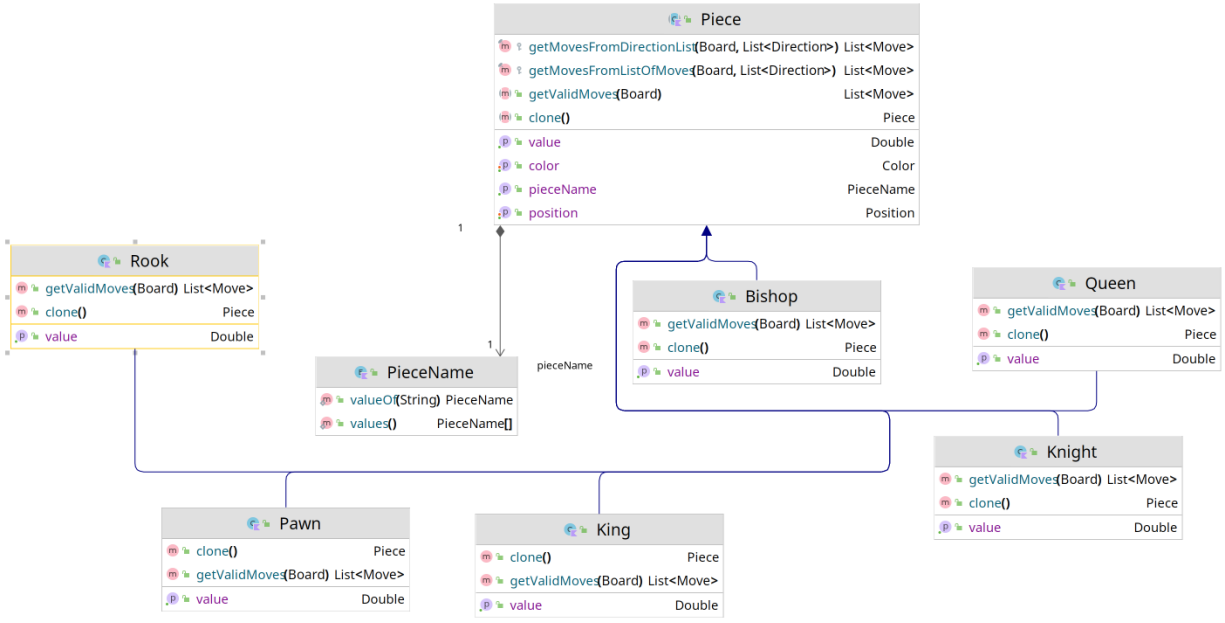

Figure 11 Pieces class diagram

Notice that there are two helper protected functions in Piece that produce moves for the other pieces by reusing a lot of the code you would typically duplicate in Rook and Queen, for example. The diagram for piece classes also displays the methods and attributes implemented.

5.2.3. Board

All moves and pieces are used to implement board functionalities. The color of the player now making their choice, the arrangement of the pieces on the board, and the movement history all affect the state of the board (or game). In order to make the backtracking operation easier to implement, the history of moves is used to determine castling rights, en-passant opportunities, and to retain the stack of moves with additional information.

The main functions the Board will implement are:

- pieceAt
    - This function takes a position and returns a nullable Piece. If there isn't a piece on the board at that position, the method returns null; otherwise, it returns the piece.

- move
    - This is a fundamental operation that implements moves by receiving a Move object and changing the Board's state to match the move's specification.
- unmove
    - This function undoes the move that is on top of the history of moves, returning the board's state to what it was before the move.
- getAllValidMoves
    - This function is crucial for creating algorithms since it returns all possible moves that every piece of the current color can make.
- getState
    - The outcome of the game is indicated by this function. If so, it indicates whether black won, white won, or a tie occurred in the game.

Of fact, the project's actual source code contains a good amount more classes and functions, but most of them are implementation details that might have been handled differently. The classes and corresponding functions described here are probably present in any engine you look into, and thorough implementation is essential to ensuring error-free execution.

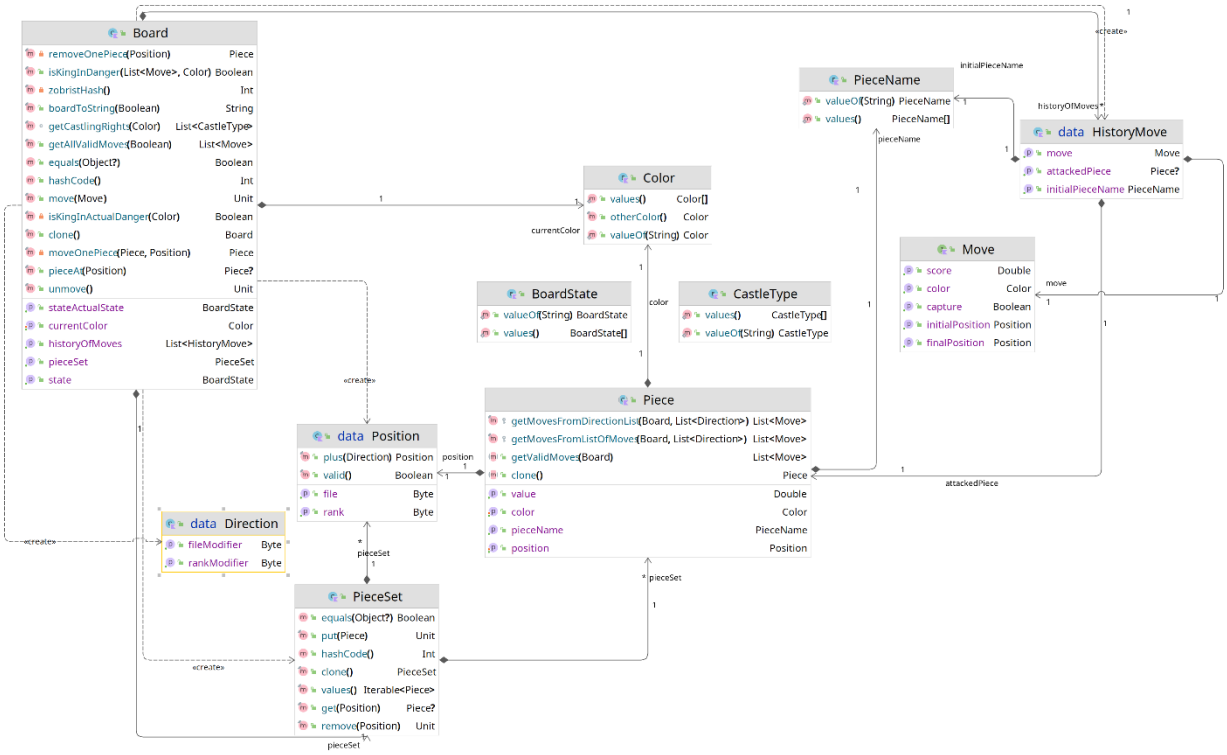

Figure 12 Board class diagram

While move is only used as an external variable and is not contained anywhere in the attributes of the classes, we can see how the pieces and the board interact in this class diagram. There are many functions in Board, but as was already said, they either support code reuse or implement support features like cloning or Zobrist hashing.

5.3. The Algorithm

This package contains just one class, Minimax, which, as its name suggests, is a min-max implementation of the suggested engine. All of the optimizations listed in chapter 3 are applied, as was already indicated, with varied degrees of effectiveness. Since comparisons will be desirable towards the conclusion of the implementation, the code is structured in a way that makes it simple to decide whether an optimization is performed or not.

The class includes two crucial functions, namely "evaluate," which, without analyzing any moves, returns the projected score of the present state of the board as a weighted sum of the pieces with positive weights for white and negative weights for black.

Although it may be a nice source for optimization, this trivial evaluation function produces good results, so I went with that instead of using a more complex criteria to determine who is more likely to win. Of course, additional knowledge like pawn structure, piece synergy, protection of the king, and so on could be useful. However, it ultimately boils down to relatively chess-specific information.

"getBestMove," which is the second function implemented, is probably the most worthwhile and significant function in the entire project. Its fundamental algorithm is a stopping condition-based min-max algorithm (maximum depth). It is based on the observation that White wants to maximize the ultimate result whereas Black wants to minimize it, according to the evaluation function we calculated earlier. This approach has a tight relationship to the game tree since it may be viewed as an example of dynamic programming on a small game tree by indirectly building it through recursion.

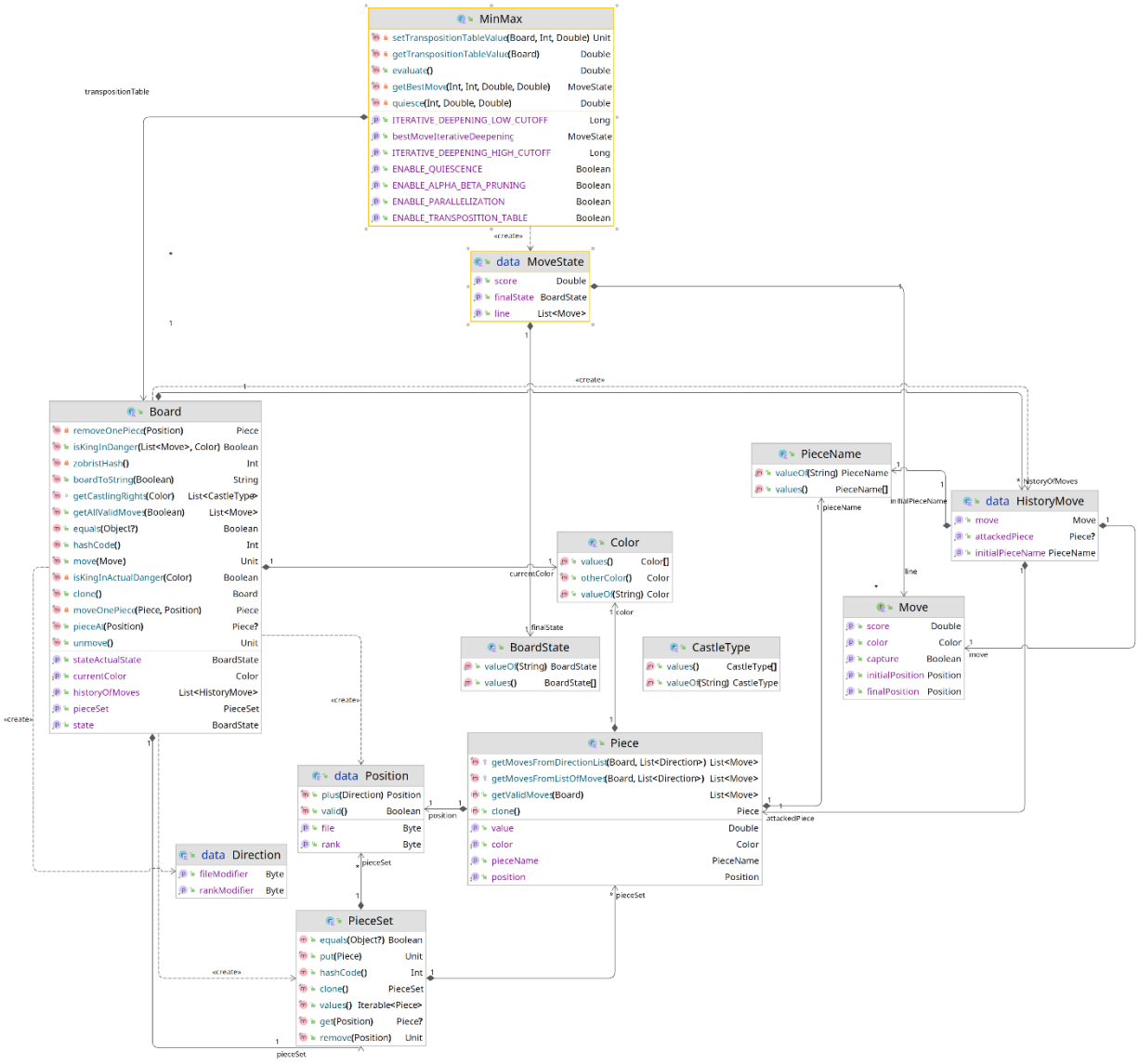

Figure 13 Minimax algorithm class diagram

The relationships between the MinMax class, the Board class, the Move class, as well as other assistance classes, may be seen in the class diagram. By adding the transposition table, which converts boards to numbers, and the currently being studied board, the MinMax class builds MoveStates, which contain Moves and multiple Boards.

The following information relates to the application of each approach specified in chapter 3, together with any relevant customizable settings.

5.3.1. Alpha-Beta Pruning

The pseudocode from subsection 3.2 was simply implemented using alpha-beta pruning. Given that the quiescence method dominates the runtime, it is noteworthy that Alpha-Beta was also utilized there. Alpha-beta doesn't have any programmable parameters.

5.3.2. Quiescence

Quiescence was implemented in the exact same way as the standard minimax, with the exception that it filters out all movements other than captures to only take into account captures. The variable parameter is the greatest depth; it may be unbounded or restricted to a small integer. Going for a small number results in poor returns from quiescence and was discovered to encourage bad moves, but going for a very large number (effectively unlimited) also resulted in poor moves because the depth of the actual minimax algorithm was stymied by the extremely high number of quiescent nodes. In practice, a sweet spot was discovered to be 10.

5.3.3. Transposition Table

This algorithm's construction of the transposition table was similarly straightforward but extremely efficient in cutting down on runtime (an observed x2 reduction in overall time spent). I simply decided to use the Zobrist hash to cache each state I came across, keeping the map size to a maximum of 1 million, and checking whether the state was already present in the table at the start of the recursion. Because states assessed with depth=1 might produce useless results at deeper depths, extra care had to be taken to ensure that the hash table was reset before each run of the iterative deepening operation.

The table's size and the maximum depth to which I can update it are changeable factors, but it was discovered that being lenient with both of these was best (that is large values).

5.3.4. Parallelization

I indicated earlier that despite using parallelization in this approach, I was unable to get adequate results. Starting so many threads resulted in an excessive amount of overhead, negating any benefits from using several CPUs. Additionally, extra subtrees that were not required were frequently computed, which extended the overall computation time. The number

of threads spawned by each node and the depth cutoff after which no threads would be produced are the only two modifiable factors.

5.4. Server

It is a given that there must be a means to expose the methods so that a chess program may interact with them since the application was separated between a backend and a frontend project. The server was created using Spring Boot, a relatively straightforward and well-known programming technology that can be utilized with Kotlin. The server has made the following methods available:

- 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 starts the minimax algorithm after receiving the game identification. It then calculates the optimal move.
  - After being saved in the backend's memory, it returns the new board with the move played on it.
- move
  - This method returns the board with the move applied to it after receiving a move and a game identification.
  - It eliminates code duplication to prevent the frontend from implementing the same feature and uses the rigorously tested backend code.
- getMoves
  - When a game identification is provided, this method returns all the legal moves that can be made.

- The user will be shown the moves they are permitted to make based on the returned list when the client calls this method.

Just enough of this relatively small set of operations is available to play against the algorithm and try various moves. Since the emphasis is on algorithm performance rather than user experience, no other Spring Boot features are used, yet the user experience is still good.

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

Since data must be transmitted from the backend to the frontend and vice versa, some description of the contents of the objects is necessary. The data-transfer objects (DTOs) that are used in both the frontend and the backend are listed below:

- Board
  - The board is the obvious choice; it displays the piece list, the current color (indicating whether white or black is to move), the history of movements, and the current board state (unfinished, draw, white win or black win).
- ExecuteMove
  - The board, the actual move, and, in the case of pawn promotion, the choice of piece promotion, are all submitted via this class to the Move operation in the backend.
- HistoryMove

    - This class is designed to be used solely within the Board and to communicate the same object that was received back to the backend.
- Move
    - The same fields that a move contains in the backend are present here as well because moves are frequently used across calls: position at the start, ending, color, and class name (to translate from base interface to specialized move such as CastlingMove)
- Piece
    - The piece includes a position, color, and class name, just like in the backend, which helps the algorithm specialize the class that gets processed.
- Position
    - The position simply consists of the rank and file and is utilized as a component of other DTOs.
- 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

The 6 components of the application are described in this subsection along with how they are conceptually implemented and used.

5.5.2.1. Chessboard

This is the primary component of the application, and it includes all of the functionality. In fact, it is a div that has 64 subcomponents, one for each square, and arranges its subitems in an 8x8 grid. It incorporates clicking capabilities so that when a piece is clicked, all potential destinations are highlighted, and selecting one of them causes the piece to be moved to that location. The move is then processed by communicating with the backend, which also instructs the engine to start its own move computation and return the new board.

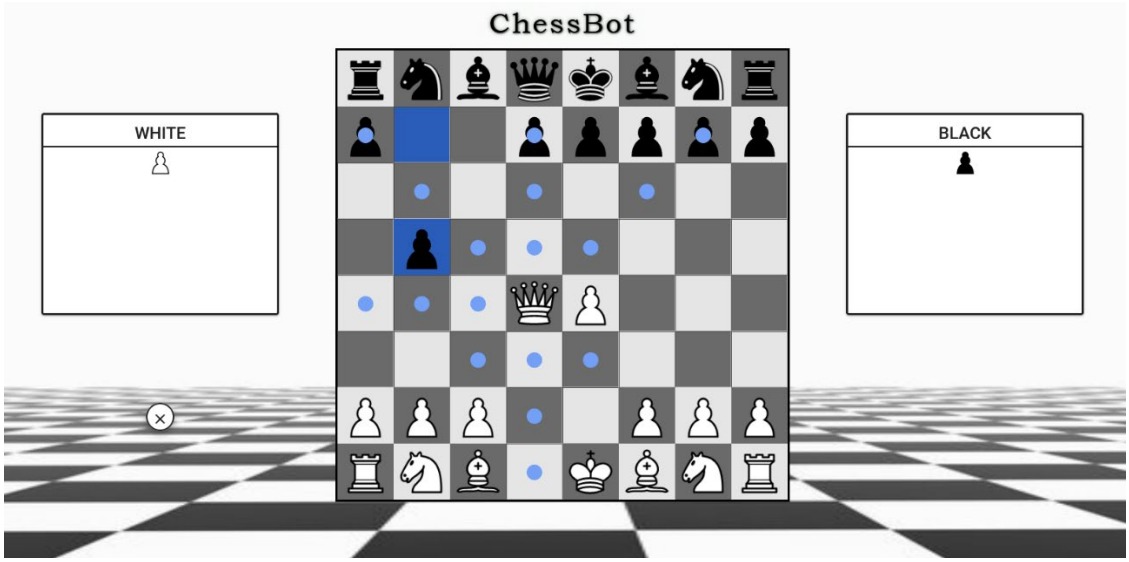

Figure 14 The chessboard component in the game page

5.5.2.2. Chess Piece

This element is a single square on the 8x8 grid that has a colored background (either light or dark), perhaps a piece, and perhaps a green dot denoting a potential move. If this square was a part of the most recent move made on the board, a blue background might also be visible. The component is given a place, a piece, and a flag as input, with the flag designating whether or not the piece is allowed to travel there. It then just parses these inputs to give the user the appropriate-looking square.

5.5.2.3. Promotion Choice

This component is used to give the player the option of selecting which piece they want to promote their pawn into. The Chessboard component opens it as a dialog, and clicking a piece will send the decision back to the chessboard component, where it will then complete the information needed for the move.

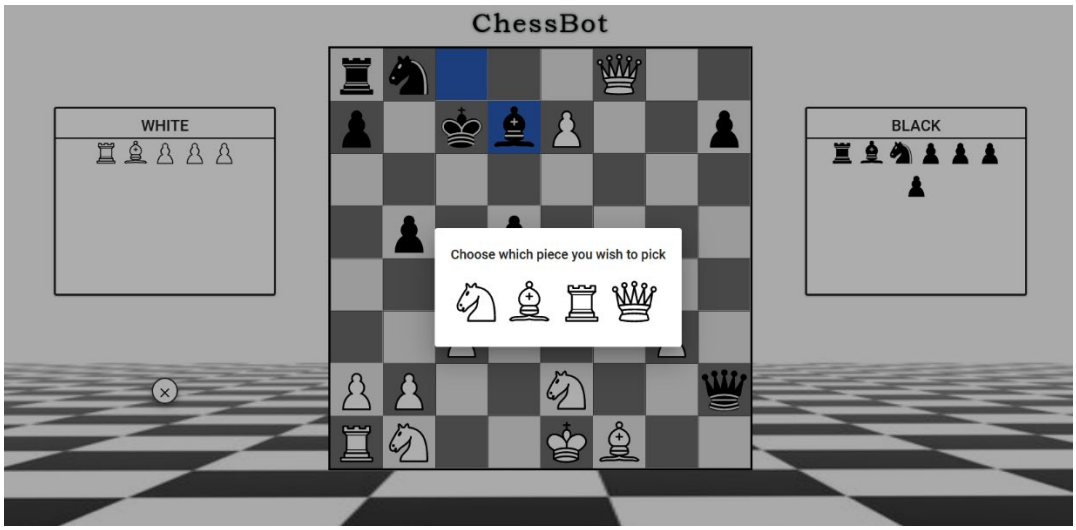

Figure 15 The pawn promotion

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.

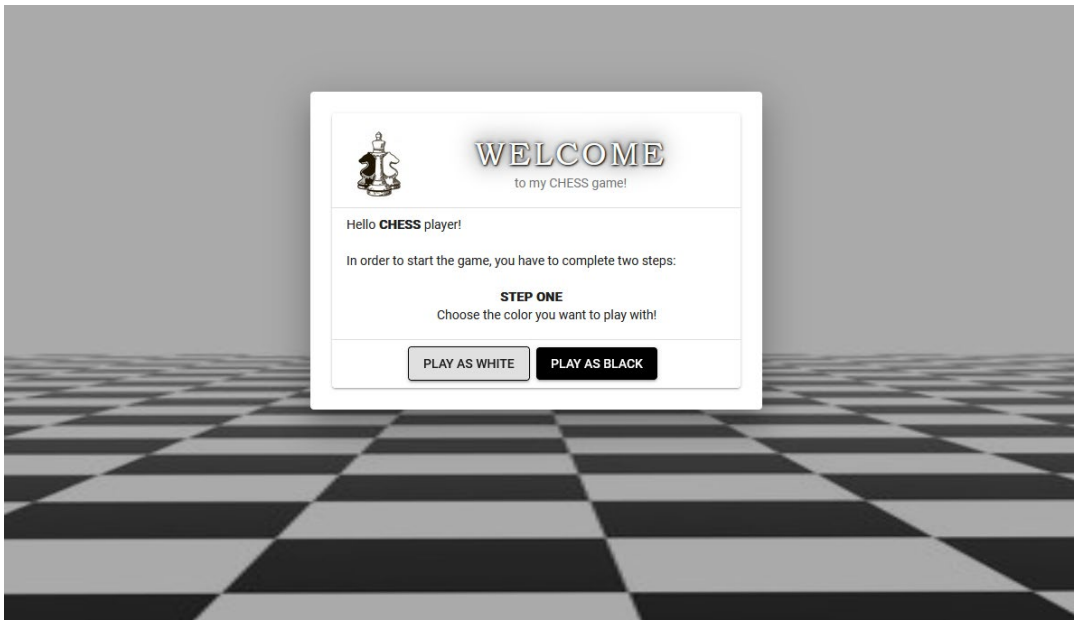

Figure 16 The color dialog

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).

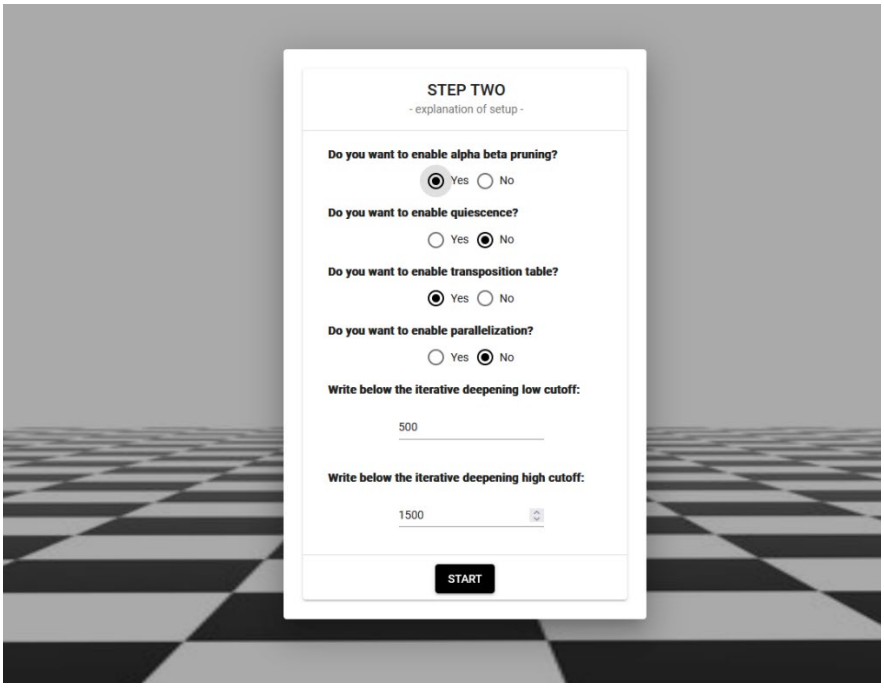

Figure 17 The setup component

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

While it is difficult to definitively say that all of the techniques improved play, there are certainly situations for each that suit them. Quiescence search makes analysis difficult, but on first glance the moves played by the engine were good, even with a small time cutoff. Parallelism

as implemented only improved runtime in combinations with some techniques, but failed to push the best times on the initial board. Alpha-beta pruning has been able to lower runtime by a factor of about 4, which is to be expected because the majority of moves are terrible and not taking them into consideration is quite advantageous. The transposition table also produced runtime speed-up by a factor of more than 2, emphasizing once more how frequently chess positions repeat. The heuristics that optimize move ordering also significantly contributed to alpha-beta's speed.

This engine, in my opinion, can win against inexperienced players, defend itself from immediate dangers, and even be entertaining to play against and try to outwit. There were advantages and downsides to writing such a program in a high-level object-oriented programming language. The key advantage was simplicity of implementation due to OOP abstractions that allowed for simple code reuse, distinct hierarchies and design patterns, and consistency in using pre-existing data structures like maps and arrays of any size. Writing a chess engine in Kotlin as a learning exercise is, in my opinion, a good opportunity and may allow one to acquire abilities they otherwise wouldn't have, as using such a potent language to implement a challenging algorithm will emphasize the language's advantages. I would argue that the effort has been successful despite the language's high-level overhead, because a suitable competitor against inexperienced players was produced as a result of a few strategies applied to a straightforward minimax algorithm.

6.4. Further Work

The chess algorithm used in this thesis produced encouraging results, but more work could potentially greatly raise the level of play. The inefficiency of utilizing only one CPU to do the operation would finally be overcome with more research into more effective parallelization, and depending on the device's number of CPU cores, the execution of the program would be sped up several times.

The heuristics used were also subpar, and I've actually just implemented simple static evaluation and move ordering criteria. The outcomes would be greatly enhanced by more

aggressive pruning, better evaluation that takes positional advantage alongside material advantage, opening and endgame tables.

Finding a way to use more cutting-edge artificial intelligence techniques, such neural networks, could also be a road to achieving maximum efficiency, as we've seen with the state-of-the-art of chess engines. To determine the optimal parameters you can set, you may also increase the amount of customizable parameters and run evolutionary algorithms based on them to compare different game engines.

Overall, a lot of research might improve the program's outcomes, and while the early findings are not groundbreaking, they are also not terrible. If given additional time, the program would begin moving up the ELO ladder and be equally as successful in defeating master players as Stockfish or Leela Chess.

7. 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.