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

DIPLOMA THESIS

Evaluating the performance of 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

Evaluating the performance of a chess minimax algorithm implemented in an object-oriented language

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 attempt to recreate the success of other chess engines 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 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 fifth 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[2] beat reigning world champion Garry Kasparov). Naturally, over time chess engines have only gotten better, in fact so good that even evaluating the top performers is a daunting task, as there is no human benchmark anywhere near their level. One reason the engines have gotten so good over time is that the game is played between just two players, involving no hidden information or randomness. This makes chess a prime opportunity for computers to tackle playing, the reason being that we can employ more rigorous techniques than heuristics, or even combine the two in order to achieve peak performance. Indeed, many ideas, optimizations, tricks and heuristics have been developed during these last few decades, ideas 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 second chapter, each of which are implemented by the engine. The third chapter will describe the practical part of the thesis, the structure of the frontend side and for the backend. The fourth chapter describes the observed performance of various iterations of the algorithm, analysis of

games played against it and lastly patterns observed in the engine's play. Finally, the conclusions of the thesis are drawn, reiterating the key take-aways and the meaning behind results.

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[7] 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[1], a ground-breaking neural network algorithm that after 4 hours of training matched the best algorithm to that date, Stockfish, and in 9 hours of training it was able to defeat Stockfish in 28 games out of 100, drawing the remainder.

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

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

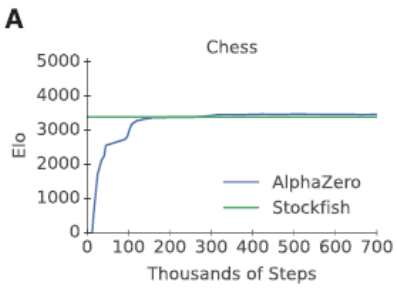

Fig. 1 Evolution of Elo rating for AlphaZero[1]

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

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

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

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

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[3]. After playing 500 million games by itself it achieves performance that is comparable to that of Stockfish. Leela does not however run on a super computer like AlphaZero, but relies on the community to run games locally and improve itself from self-play among many volunteers.

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

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

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.

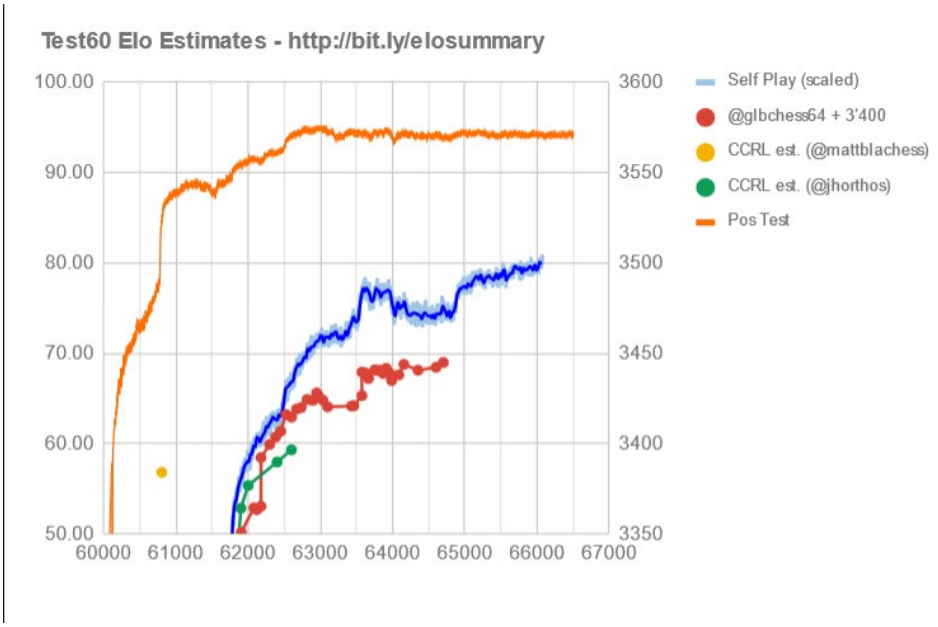

Fig. 2 The evolution of Leela's Elo by various metrics (28-03-2022)[3]

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[8] 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. Because of this, 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, depth, maximizingPlayer)
    if depth = 0 or node is a terminal node
        return the heuristic value of node

    if maximizingPlayer
        bestValue := −∞
        for each child of node
            v := minimax(child, depth − 1, FALSE)
            bestValue := max(bestValue, v)
        return bestValue

    else    (* minimizing player *)
        bestValue := +∞
        for each child of node
            v := minimax(child, depth − 1, TRUE)
            bestValue := min(bestValue, v)
        return bestValue
```

Fig. 3 Minimax pseudocode[8]

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[9]. 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.

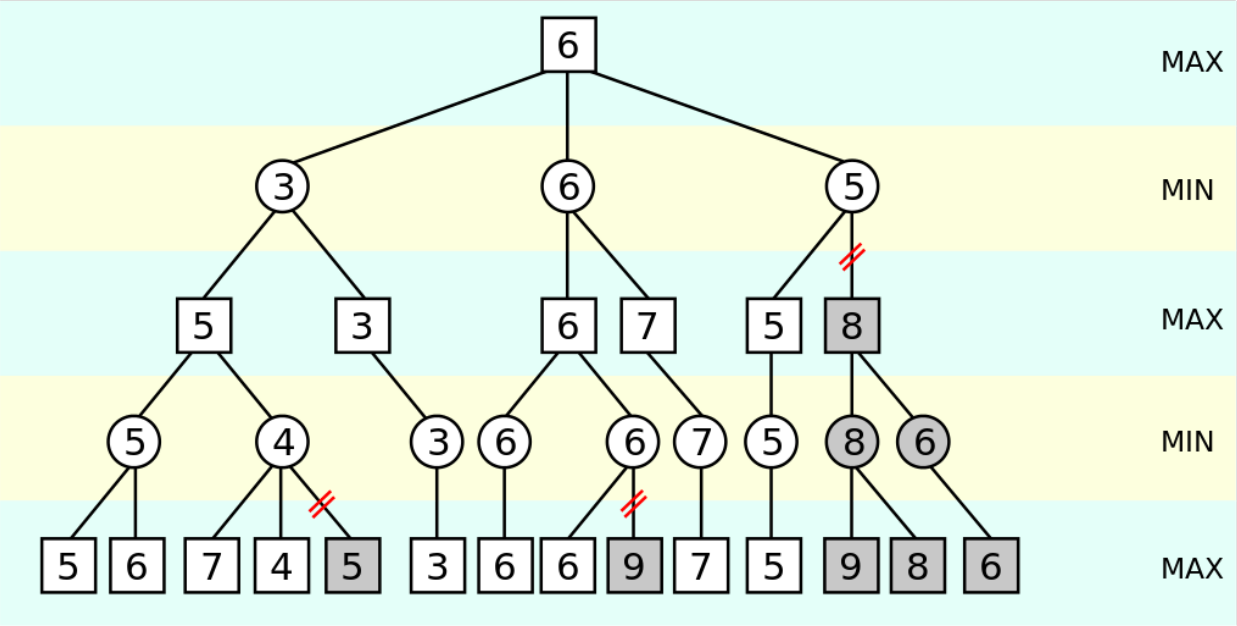

Fig. 4 Alpa-Beta Pruning demonstration[9]

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

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[10] 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.

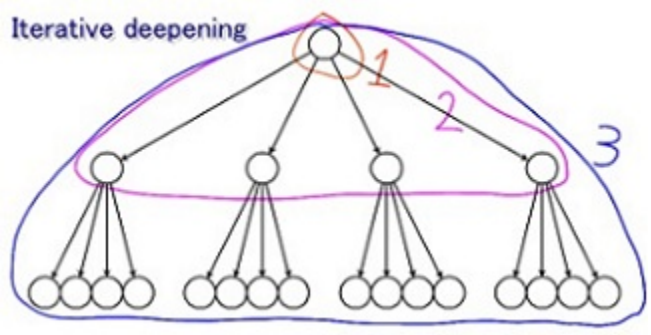

Fig. 5 Iterative deepening example[10]

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[11] is so important is that when I evaluate a board, I assume the board is "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 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 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 much 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. In our experiments quiescent nodes make up between 50% and 99% of the nodes processed, and although it may seem like the extra time spent is not worth, it still improves the quality of moves over the alternative since one depth less is not as bad when you have good evaluation for leaf nodes.

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 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[13]. 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.

```
constant indices
    white_pawn := 1
    white_rook := 2
    # etc.
    black_king := 12

function init_zobrist():
    # fill a table of random numbers/bitstrings
    table := a 2-d array of size 64×12
    for i from 1 to 64:  # loop over the board, represented as a linear array
        for j from 1 to 12:      # loop over the pieces
            table[i][j] := random_bitstring()
    table.black_to_move = random_bitstring()

function hash(board):
    h := 0
    if is_black_turn(board):
        h := h XOR table.black_to_move
    for i from 1 to 64:      # loop over the board positions
        if board[i] ≠ empty:
            j := the piece at board[i], as listed in the constant indices, above
            h := h XOR table[i][j]
    return h
```

Fig. 6 Zobrist hash pseudocode[13]

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[12] 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.

Unfortunately, experiments did not yield great results, even playing with the depth cutoffs for applying the technique I found that the time of the execution was either longer or equal to the fully sequential case.

There are ways to implement parallelization even for alpha-beta algorithms, the Young Brothers Wait Concept[15] 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[14], that is I prioritize moves that attack pieces that are more valuable than the piece attacked. 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. Engine implementation

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

4.1. 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[5]. 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.

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

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 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:

- 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.

4.2. 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.

4.2.1. 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.

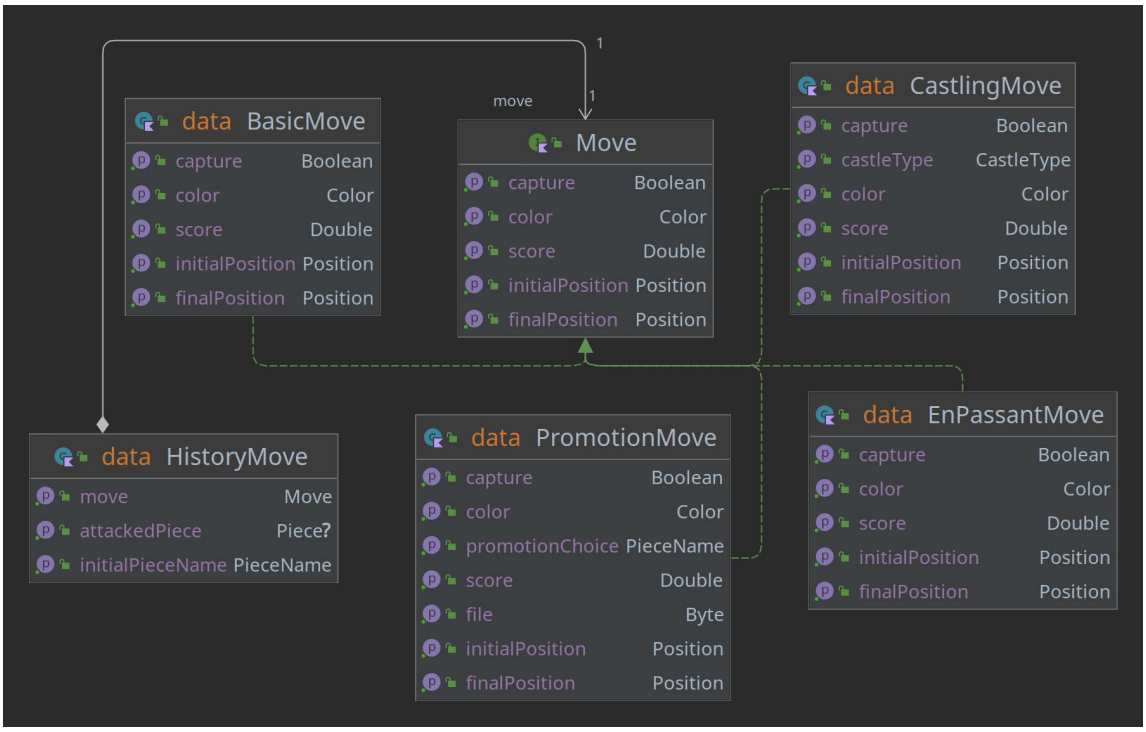

Fig. 7 The class diagram for the moves

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.

4.2.2. 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.

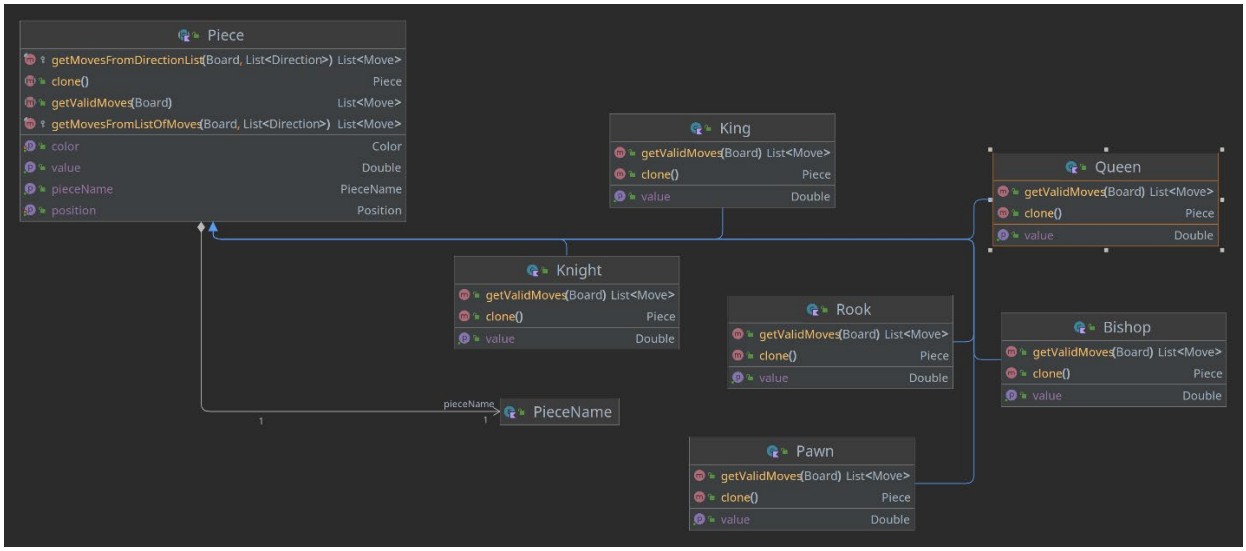

Fig. 8 Pieces class diagram

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.

4.2.3. 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.

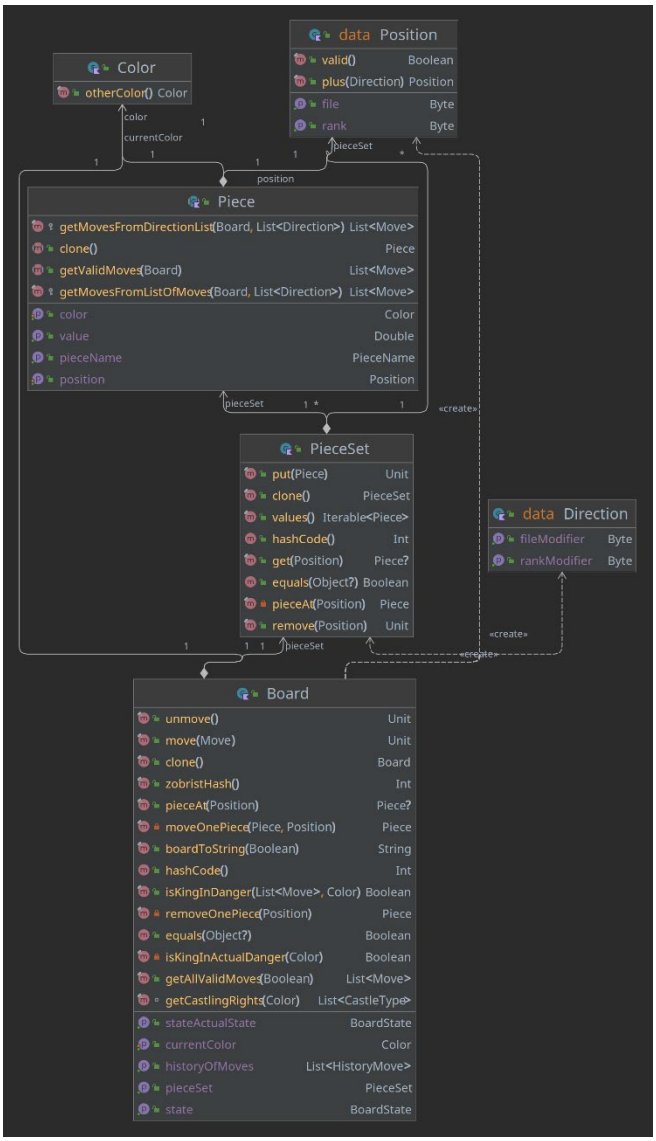

Fig. 9 Board class diagram

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.

4.3. 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.

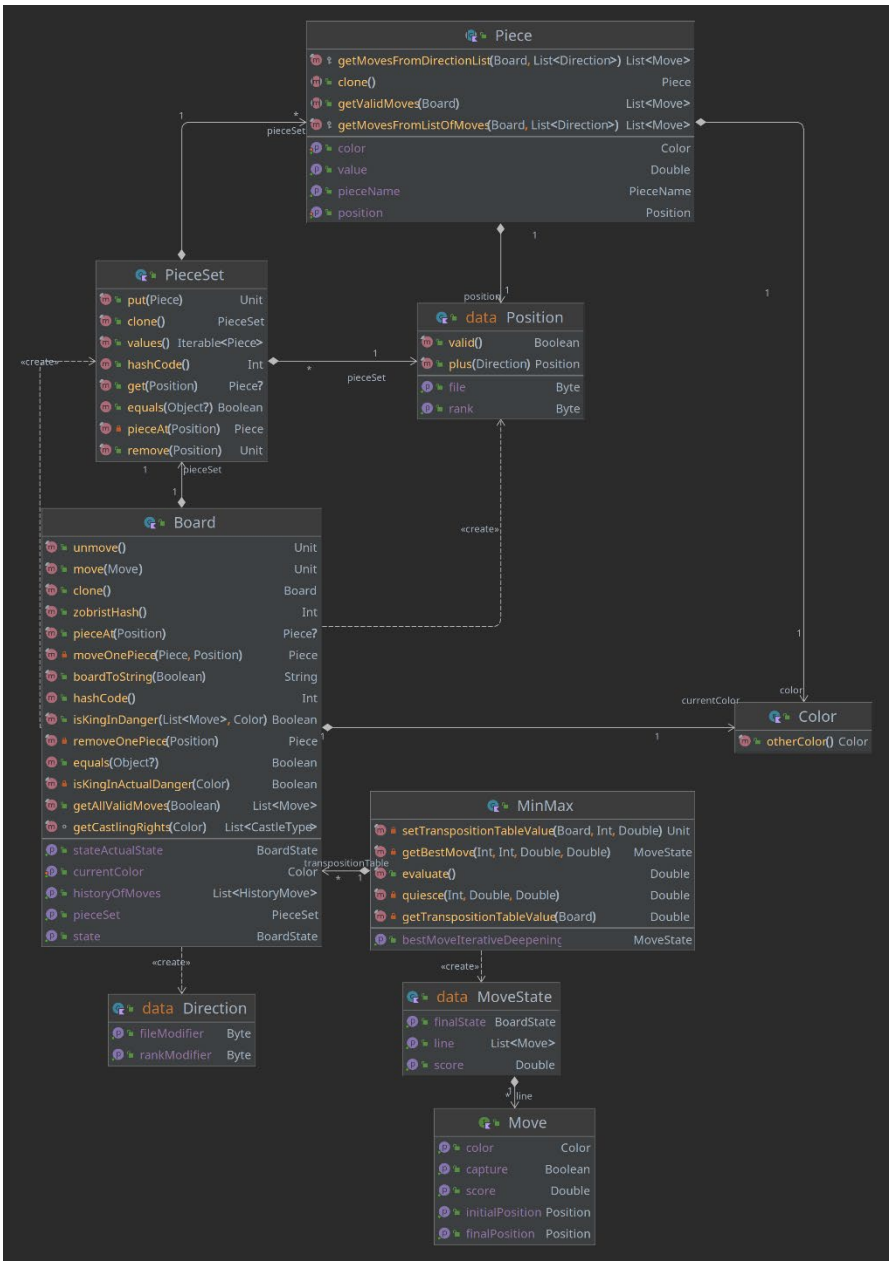

Fig. 10 Minimax algorithm class diagram

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.

4.3.1. 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.

4.3.2. 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.

4.3.3. 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).

4.3.4. 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.

4.4. 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:

- getStartingBoard
  - This method returns to the client a board with the starting formation.
  - It is typically called by the client to initialize a new game.
- computeMove
  - This method receives a board and starts the minimax algorithm to compute the best move. It returns the new board with the move played on it.
- Move
  - This method submits a move and a board 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 state of a board. 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.

4.5. Frontend

A simple client written in Angular[4], the popular Typescript framework has also been provided to easily interact with the engine. There is only one page, and it contains just the chessboard. There is also a service class that makes http calls to the backend which were outlined in subsection 4.4.

4.5.1. 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.

4.5.2. Components

This subsection presents the only 3 components of the application, how they are conceptually implemented and used.

4.5.2.1. 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.

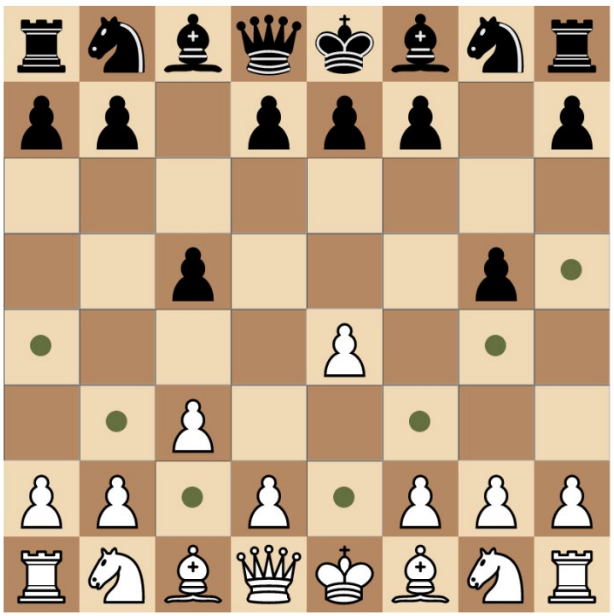

Fig. 11 The chessboard component

4.5.2.2. 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 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.

4.5.2.3. 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.

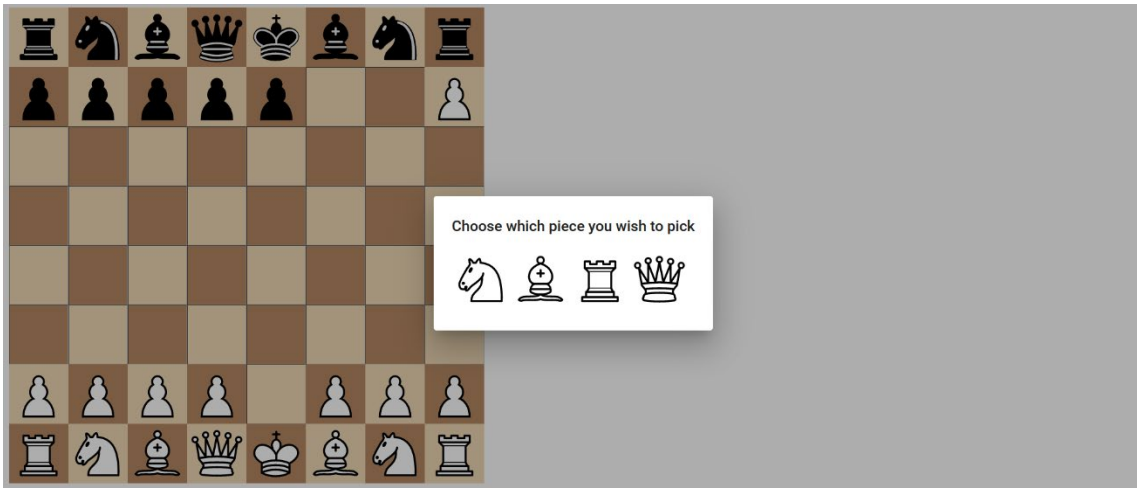

Fig. 12 The pawn promotion

5. Results and Conclusions

5.1. Game 1

PGN: 1. e4 c5 2. Nf3 g6 3. c3 f5 4. exf5 g5 5. Nxg5 h5 6. Bd3 a5 7. O-O e5 8. Re1 Qe7 9. h4 d5 10. Qa4+ b5 11. Bxb5+ Bd7 12. Bxd7+ Qxd7 13. Rxe5+ Ne7 14. Qb3 Qxf5 15. Rxf5 Nbc6 16. Rxd5 Rh6 17. Rxc5 a4 18. Qf7+ Kd8 19. Qxf8+ Kc7 20. Qxh6 Ra5 21. Ne6+ Kb8 22. Qh8+ Nd8 23. Qxd8+ Ka7 24. Rxa5+ Kb7 25. Qc7#

This game was played with quiescence up to depth 10 and no parallelization. Often, especially in the early game due to predominant quiescence it only managed to analyze up to depth 2 or 3, which has led to pretty huge blunders from the algorithm. Huge mistakes early in the game lead to a large material advantage on my side and the game was ultimately a breeze.

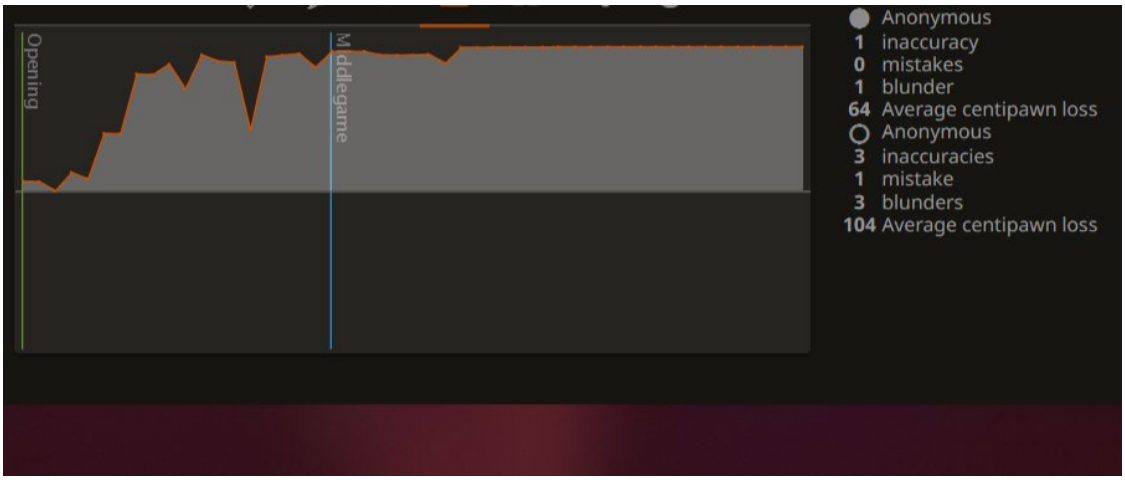

Fig. 13 Analysis of the game from stockfish performed on lichess.org

The implementation of quiescence in this algorithm proved to not be as fruitful as expected, experimenting with the engine made it pretty clear that quiescence yields very mad moves sometimes, which in chess you cannot afford to make. Gaining a considerable material advantage is enough to make the win easy, thus this game proves that quiescence as implemented only worsens the quality of the game.

5.2. Game 2

PGN: 1. e4 e5 2. Nf3 Nf6 3. Nc3 Bd6 4. d4 Ng8 5. dxe5 Be7 6. Bc4 Bb4 7. O-O h6 8. Bf4 g6 9. Nd5 Bf8 10. Qd4 Nc6 11. Qc3 Bg7 12. Nd4 Nxe5 13. Bb3 c6 14. Bxe5 Bxe5 15. Ne3 b5 16. Qc5 Qc7 17.

g3 Ne7 18. f4 Bxd4 19. Qxd4 Rf8 20. Qg7 Qb6 21. Rf3 Qc5 22. Qxh6 Qd4 23. c3 Rh8 24. Qg5 Qg7 25. f5 Qh7 26. Rf2 Qg8 27. fxg6 Qxg6 28. Bxf7+ Qxf7 29. Rxf7 Kxf7 30. Rf1+ Ke8 31. Qe5 b4 32. Qxh8+ Ng8 33. Qxg8+ Ke7 34. Qf7+ Kd8 35. cxb4 Rb8 36. Qf8+ Kc7 37. e5 Rb5 38. Qd6+ Kb7 39. a3 a6 40. Rc1 Kb6 41. Nc4+ Kb7 42. Rc3 Ka7 43. Qc7+ Bb7 44. Nd6 Ka8 45. Qxd7 Kb8 46. Nxb5 axb5 47. e6 Bc8 48. Qxc6 Bxe6 49. Qxe6 Kb7 50. Rc6 Ka7 51. Qd7+ Ka8 52. Rc8#

This game was played without quiescence and without parallelization. Because quiescence no longer dominated the game tree, the algorithm was able to go as deep as 5 and 6 moves in the tree early in the game. This proved to be a much better predictor of moves since even with the best heuristics possible, a depth of 2 will inevitably return a bad result at some point. I still managed to win this game as well, but the bot put up much more of a fight and I was unsure of the victor for most of the game.

Fig. 14 Analysis of the game from stockfish performed on lichess.org

Note that I was able to produce a great advantage early in the game, the reason being that the bot only considers material in evaluating a position, and without the possibility of easy captures in the beginning it will think of it as mostly equal. Because of this I gain positional advantage which secures and evolves my pieces, setting myself up for a smooth game. Inexperience leads me to lose much of the advantage, even reaching a drawn position, but I'm soon able to outsmart the algorithm yet again and by paying attention to possible captures I can keep the advantage for the remainder of the game. Finally, I mate at move 52, but I still consider the algorithm reasonably competent.

5.3. Game 3

PGN: 1. e4 e6 2. d4 Qh4 3. Nc3 Bb4 4. Be3 Qxe4 5. Qf3 Qxc2 6. Rc1 Qxb2 7. Ne2 Qxa2 8. g3 Qa5 9. Bg2 Bf8 10. O-O Qa6 11. d5 Ba3 12. Ra1 Nf6 13. Bc1 e5 14. Bxa3 Qc4 15. d6 Na6 16. dxc7 Nxc7 17. Rfc1 Ne6 18. Qe3 Qb3 19. Qxe5 Qc4 20. Nf4 Qa6 21. Nxe6 dxe6 22. Nd5 Nxd5 23. Qxg7 Qb5 24. Qxh8+ Kd7 25. Bxd5 Qa4 26. Bc4 b5 27. Rd1+ Kc7 28. Qd8+ Kc6 29. Qd6+ Kb7 30. Bxb5 Qb3 31. Qc6+ Kb8 32. Bd6#

Game 3 was played without quiescence and with parallelism of two threads per node for the first two levels of the game tree. This game marked the first time the engine gained an advantage, remarkably early at that. It still managed to lose it and couldn't bounce back, as even depth of 5 moves is still not enough to be immune to simple tactics.

Fig. 15 Analysis of the game from stockfish performed on lichess.org

Ultimately, I believe the current implementation of parallelism had no positive impact on the performance of the engine, the maximum depth reached was a mere 7 compared to the previous game's 10. Also, the observed quality of the moves was somewhat similar, the algorithm was aggressive in the beginning and captured several pawns, but then failed to defend itself.

5.4. Conclusions

Unfortunately, two of the 7 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 was not able to produce any benefits in runtime. Alpha-beta pruning has managed to reduce runtime by a factor of between 2 and 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 factor up to 2, highlighting yet again how often positions repeat in chess. The runtime speedup was measured by spinning up the algorithm on a starting board and observing how quickly the last depth iteration has finished.

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 reusage, 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.

5.5. 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.

6. Bibliography

[1] Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T. and Lillicrap, T., 2018. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science*, *362*(6419), pp.1140-1144.

[2] Campbell, M., Hoane Jr, A.J. and Hsu, F.H., 2002. Deep blue. *Artificial intelligence*, *134*(1-2), pp.57-83.

[3] https://training.lczero.org/

[4] https://angular.io/

[5] https://spring.io/

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

[7] https://en.wikipedia.org/wiki/Elo_rating_system

[8] https://en.wikipedia.org/wiki/Minimax

[9] https://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning

[10] https://www.chessprogramming.org/Iterative_Deepening

[11] https://www.chessprogramming.org/Quiescence_Search

[12] https://www.chessprogramming.org/Transposition_Table

[13] https://www.chessprogramming.org/Zobrist_Hashing

[14] https://www.chessprogramming.org/MVV-LVA

[15] https://www.chessprogramming.org/Young_Brothers_Wait_Concept