Tampereen yliopisto

Igor Podsechin

PROGRAMMING A CHESS ENGINE IN C++

Bachelor's thesis
Faculty of Information Technology and Communication Sciences
Examiner: Mikko Nurminen
January 2022

TABLE OF CONTENTS

1. INTRODUCTION

Chess is one of the world's oldest board games and quite possibly the most popular one, having been played by people for many centuries. The reason for this is perhaps because the game has simple and easy-to-learn rules, but at the same time it offers deep strategic and analytical possibilities. It therefore comes as no surprise that the scientific community has for a long time tried to create a machine that is capable of playing chess independently. This bachelor thesis' research question could be summarized by the following question: "How can a computer be programmed to play chess?".

The purpose of this paper is to go through step by step on how to write a chess engine using the C++ programming language in a clear and straightforward fashion. The resulting program should be capable of playing the game intelligently or at least give off such an impression. After all, the only thing a computer can really do is execute simple, predetermined instructions extremely quickly. Giving the right instructions in the correct order creates the illusion of a smart and calculating chess player, which of course is not true in reality.

The research problem can be broken down into smaller parts, out of which each has an important role when writing a chess engine. These are for example the playing board and how to model the pieces on it, defining the ruleset correctly and deciding on what basis a certain move could be classified as a good or a bad one. The paper also considers the research problem from the performance point of view to some extent, but the main focus will stay on the implementation side of things. In an ideal situation a person who has experience with programming and knows the rules of chess could read through the text and get a general overview of how a traditional chess engine operates.

The structure of the paper is as follows. First, the history and development of computer chess will be covered briefly. Then one possible way of modelling the chess board and how the pieces move will be presented. After this the method of selecting which move to make will be described. Finally the resulting chess engine will be examined, possible improvements will be discussed and a conclusion will be presented.

2. HISTORY OF COMPUTER CHESS

The term computer chess in general refers to the cooperation of computer hardware and software with the purpose of playing chess without the help of human guidance. [1] The idea of a machine capable of playing chess has existed as early as the 18th century, when an Austrian inventor by the name of Wolfgang von Kempelen created a mechanical chess automaton which impressed audiences around Europe. However, as it turns out the device was in reality a hoax, as the moves were actually chosen by a small-sized man hidden inside it. [2]

Actual development of computer chess began when the first computers were built in the late 1940s. In 1950 Claude Shannon, who is widely regarded as “the father of information theory”, published the scientific article "Programming a Computer for Playing Chess " which was one of the first papers to outline the algorithmic methods of the game. During the following year Alan Turing wrote a program with a pen and paper that was theoretically able to play a whole match of chess. However it wasn't until 1956 that the performance of early computers improved enough that American scientists Paul Stein and Mark Wells were able to write a program for the MANIAC 1 -computer, which was originally used to calculate nuclear reactions. The engine could play a rudimentary version of chess with a smaller 6×6 board and a simplified set of rules. Running at 11 000 operations per second and with the use of an algorithm called minimax it was able to calculate four moves ahead, finding the move it thought best after about twelve minutes. [3]

In the next few years chess programs were already able to play using a full 8×8 board and could also utilize a more advanced search algorithm called alpha-beta pruning. Despite this it would take many years for them to play at the same level as human players. The issue was that the lack of computer performance meant that engines could not calculate very far ahead and were therefore prone to making mistakes that could be exploited by skilled players. As a result there was some research into search algorithms that attempted to be more selective when examining its next move, but by the 1970s the increase in computing power ultimately led to the fact that an exhaustive brute-force style search which examined all possible moves became more popular.

3. MODELING THE GAME OF CHESS

Before a computer can start playing chess, the game must first be described in such a way that it understands it. Concepts like the gameboard and pieces have to be mathematically defined and this also includes more abstract ideas such as what makes a move legal or what a move even is. There are of course many ways of doing this and this chapter will present one fairly straightforward representation that was chosen when writing the chess engine for this paper. However, before this a few important properties related to the game should be discussed briefly.

3.1 Properties of chess

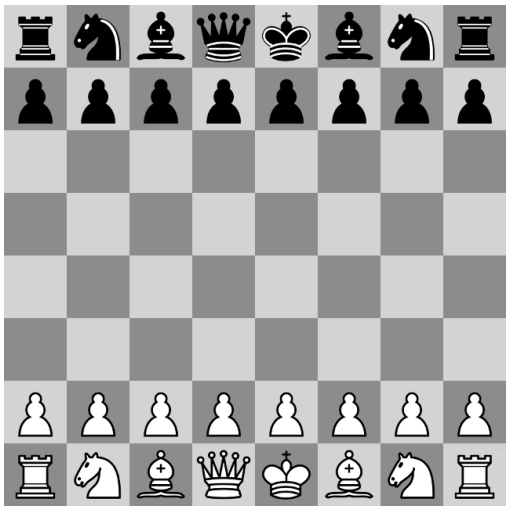
Chess can be classified as a perfect information, turn-based, zero-sum game between two players. [7] Perfect information means that both players have full knowledge of the current state of the game and all the events that have previously taken place. Zero-sum on the other hand means that any gain in advantage that one player makes is equivalent to the other player's loss. This holds true at the end of the game (if one player wins and then the other has to lose) and also during the game. In addition, chess has no hidden information present, whereas for example in poker the players do not know their opponents cards and have to make decisions with that in mind.

Being turn-based brings up the first property that has to be modelled for the program. Since the player whose turn it is to move can either be white or black, a single true or false boolean value would be sufficient to store this data, but for the sake of clarity and readability a C++ type `enum` called `side {WHITE, BLACK}` is defined. [8] The enum is convenient as it can also be reused in other parts of the program. For instance in the function that checks if a player has lost, we can pass an argument with the `side` type to specify which player is being examined for checkmate.

Other properties of chess include the fact that there is no randomness involved and that the game is symmetric. This essentially means that the starting position on the board is mirrored along the central x-axis with both players having the exact same pieces that move the same way regardless of color. The symmetry is a property which can be taken advantage of when deciding how to represent the pieces and evaluating a position on the board, which will be described in more detail later.

3.2 Board representation

The game takes place on an 8×8 board consisting of 32 light and 32 dark squares arranged in a checkered pattern. Interestingly enough, from the perspective of the computer program having two different colored squares has no significance on the game and was most likely added to aid human visualization. As a result there is no need to encode this information anywhere. The most straightforward way of modelling the gameboard is probably by using a matrix where each element represents a single square. This can be implemented in C++ by using a two-dimensional array and is the approach that was used when writing the engine for this paper. The next question is then what data type should the array values hold?



```
char board[8][8] =
  { {-4, -2, -3, -5, -6, -3, -2, -4},
    {-1, -1, -1, -1, -1, -1, -1, -1},
    { 0,  0,  0,  0,  0,  0,  0,  0},
    { 0,  0,  0,  0,  0,  0,  0,  0},
    { 0,  0,  0,  0,  0,  0,  0,  0},
    { 0,  0,  0,  0,  0,  0,  0,  0},
    { 1,  1,  1,  1,  1,  1,  1,  1},
    { 4,  2,  3,  5,  6,  3,  2,  4} };
```

Picture 1. *A chess board in the starting position and its array representation on the right.*

Well we know that each square must hold the information of whether it is empty or occupied by a piece. As there are six unique pieces (pawn, knight, bishop, rook, queen and king) and they can either be black or white it follows that the total number of different values required is thirteen including the empty square. The smallest C++ data type char, requiring one byte of memory space, has a range from -128 to 127 which is more than enough for this task.

Each piece is assigned a separate identifying number, so for instance pawns are equal to one, knights are equal to two and so forth. This then can then be used to keep track of which piece is where on the board by accessing a specific row and column of the array. A useful technique when differentiating the two players' pieces is to assign the same numerical value to a piece but flip the sign in front of it depending on the color. So for example the white bishop has the value of 3, whereas a black one

would be -3. This way many of the functions that the engine uses can be easily used for both playing sides simply by negating the value of the piece, which in turn results in more concise programming code.

In addition to the array containing the board position there are a few more variables that must be kept saved in-between moves. The first of these is castling rights. Castling is a special move during which two pieces, the king and a rook, move during the same turn past each other on the starting row. It can only be performed once and only in the case that neither piece has moved prior to this. If we are given a position where the king and rook are in their starting squares, then without knowing the previous moves it would be impossible to know if a player could legally castle. To solve this issue a boolean flag called `can_castle` is saved alongside with the position. At the beginning it is set to true, but as soon as either the king or rook moves it is changed to false for the rest of the game. This way we will always know in any future position if castling is legal by checking the value of this variable. In total there are four of these flags for each player and both castling directions.

En passant is another special move [9] that requires knowledge of the previous moves. It is a pawn capture that can only take place if the opponent's pawn moves two squares from its starting square. If a player decides to do so, then it must be played immediately during the next turn or the right to do so is lost. Similar to castling, this information of whether a pawn can be captured en passant or not must be stored in a boolean flag for each individual pawn on the board. It is set to true once, the during the turn immediately after the pawn has moved two squares and is false at all other times.

```
// holds the state of the board including castling and en passant
struct board_state
{
    char board[8][8];
    bool pawn_two_squares_black[8];
    bool pawn_two_squares_white[8];
    bool can_castle_white[2];
    bool can_castle_black[2];

};
```

Program 1. *The data structure used to hold a board position.*

To sum up, the information that must be stored represent a chess position on is the board itself, castling rights and en passant. When the engine searches for its best move all of this information must be passed on at every step. The C++ has a data type called struct, which is a suitable way of grouping this data together into one neat package.

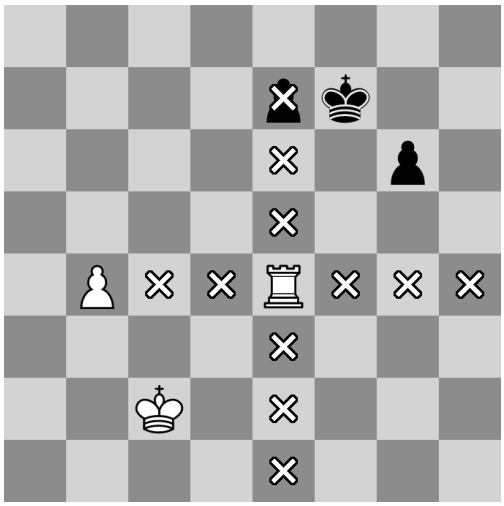
Finally, it must be noted that the two-dimensional array is not the only way of representing a position of a chess board. Bitboards are specialized bit array structures that are also capable of doing this. [10] They are more complex to implement than a simple array, but in return they provide a speed up in performance. They also contain additional information apart from the board position which can help in generating moves. Briefly explained, a bitboard stores a single boolean for each square and thus one board could for example represent all the possible squares that a piece can move to in a specific position. Correspondingly, a collection of them can be used to describe the complete position on the board. Since a total of 64 squares make up the playing board, a bitboard also fits nicely into a single 64 bit register in the processor allowing for very fast parallel calculations with multiple boards.

3.3 Implementing the rules

There are a number of different aspects to consider when implementing the rules of chess. One is the movement of pieces, which actually consists of two separate parts. If we decide to write the program in such a way that a human can play against it, then there has to be a section of code that checks if a move that was entered by is legal or not. If the answer is yes, then we carry out the move and if not then simply continue to wait for a valid one.

The second part of piece movement is similar, but from the perspective of the engine. Instead of verifying if a move is valid, it needs to generate all legal moves from a given position. This is done so it can then evaluate them and choose the best one. Both the act of checking if a move is legal and move generation have similar logic behind them, so we will cover them simultaneously.

Since each chess piece moves slightly differently, their patterns have to be programmed in case by case. For pawns it means examining if the square in front of it is unoccupied and if the answer is yes, then moving there is legal. The knight on the other hand moves in an “L” shaped pattern. In its case we check all the squares that it can reach from the starting square by moving in this pattern and if the destination square is either empty or occupied by the opponent’s piece then moving there is permitted. If the target square happens to be occupied by a friendly piece, then moving there is illegal since you cannot capture your own pieces.



Picture 2. *The rook's possible moves in this specific position marked with a cross*

For the bishop, rook and queen the method of examining possible moves is similar to the one described above. The key difference lies in the fact that because they move in straight lines, their path has to be "traced" out to see if they hit any obstacles. For example in the picture above the rook can move to all the squares that are marked with an x, including the one occupied by the black pawn, but not beyond that. To find all the legal moves a while loop can be used. The loop starts from the rook's position and iterates square by square in one direction until it hits another piece or the edge of the board. The same logic applies to the bishop and queen, except that their patterns include diagonal movement.

The king is said to be in check if it is being attacked by an opponent's piece. The king cannot make a move that places itself in check and also other pieces cannot make moves that would inadvertently place the king in check. This adds an additional step to verifying the legality of a move that any piece makes. In practice it is implemented in the following way: first each square of the game board is copied over to a temporary board and then the move is carried out on it. If the king ends up in a check after the piece is moved, then the move is deemed illegal. If not then, then the temporary board is copied back into the original board making it the current position.

The game ends if the king is checkmated, meaning that he is in check and there are no legal moves of getting out of it. Such move could be placing the king on a different square, blocking the check with another piece or capturing the attacker. The program has to examine the position for checkmate after every move, which actually adds quite a bit of computational overhead especially if there are still many pieces left on the board. Stalemate, a position where there are no legal moves, has to be examined in a similar way.

Finally, it must be mentioned that there are a few other special rules that have to be implemented, but we won't go into too much detail. These would include pawn promotion, en passant and castling. They are fairly straightforward and can be looked up from the source code directly if needed to.

Once the rules have been implemented it is very important to verify that they are correct. If there happen to be any errors then as a result it could introduce illegal moves, make legal moves impossible to perform or perhaps give a wrong outcome when examining for game ending conditions. This could lead to the engine making suboptimal moves and besides that it would not even be playing chess correctly anymore, but some other game instead.

Table 1. *Number of possible chess positions after n moves* [11]

Moves	*1*	*2*	*3*	*4*	*5*	*6*	*7*
Possible positions	20	400	8 902	197 281	4 865 609	119 060 324	3 195 901 860

Aside from manually checking each rule after its addition, a useful way to verify the correctness is to let the program play through all the legal moves turn by turn up to a certain depth and count the number of positions it generates. The table shown above can be used to demonstrate this.

At the start of the game white can move each pawn one or two squares forward ($8 \cdot 2 = 16$) and each knight can go two to different squares ($2 + 2 = 4$), so in total 20 different moves are possible. After this is black's turn and they can also move their pieces to 20 different places regardless of what white did. We can multiply these two number together to get the total number of possible positions after two turns, which is 400. Things get more complicated after this as pieces are able to capture each other, but nevertheless if the engine outputs the same number after n moves, then it is quite certain that the rules have been programmed in correctly. Online these are called perft (performance test) results [12] and they offer various different positions along with their respective move counts, which can be helpful when checking for errors.

4. CHOOSING A MOVE

This chapter will cover perhaps the most important aspect of a chess engine, which is how it chooses the move it plays in a certain position. Before that though it might be useful to briefly look at a human's thinking process while making that same decision. Although it is impossible to know the answer to this question for certain, there are some general themes that usually hold true.

4.1 Humans vs computer

Humans rely quite a lot on intuition when playing chess. This means that moves that are obviously losing, such as giving up the queen for a less valuable piece, are quickly disregarded without dedicating much thought to them. Instead, a few of the most promising moves are usually looked at in more detail. The thought process may go along the lines of: "If I make this move, then what is my opponent's best move in reply. After he or she makes that reply what is my best continuation…" and so on. Grandmasters are said to be able to calculate as far as 20 moves ahead depending on the position [13], but in most cases the number is not as high.

As it turns out, the chess engine's method of operation is not all that different. Let's for example say the computer plays as the black pieces. What it essentially does, is look at all the possible moves black can make from the current position and then from those resulting positions it looks at all the possible moves white can make and this continues up until a certain depth. The result can be viewed as a tree data structure, with the nodes being the different positions (the root node being the original position) and the edges representing the moves that were made.

In addition to this, each node is given a score using a so-called evaluation function. The number it returns is used to describe whether the position is better for either black or white. If it is zero then the position is considered equal, if it is greater than zero then it is advantageous for white and if it's less then black is better. The evaluations are used to determine the path to the most favorable leaf node from the point of view of the computer, which then can be traced back up the tree to select the best move. What was described is essentially called the minimax algorithm and it will be covered in more detail in the next section.

The key difference between the human thought process and the computer algorithm is that humans are able to utilize a lot of pattern recognition based on previous experience. So while a computer can to go through thousands of positions per second, a person is still able to spot good moves intuitively

without the need to look at every possible continuation. Another difference is that humans often tend to formulate plans while playing, such as attacking the king or occupying an important square on the board. Computers on the other hand do none of this, as they instead coldly calculate through all the possibilities using the so-called brute-force method. As a consequence, humans can sometimes have a hard time understanding the reasoning behind a computer's particular move.

4.2 Minimax algorithm

The minimax algorithm is the step by step procedure of how the search tree is traversed and the best move is chosen. The name minimax comes from the fact that for each player the algorithm tries to maximize the payoff and minimize the loss. The C++ function itself is recursive, meaning that it calls itself multiple times and carries out the result as the return value. Below is the function described in pseudocode.

```
function minimax(node, depth, maximizingPlayer)
    if depth = 0 or node is a terminal node then
        return heuristic value of the node
    if maximizingPlayer then
        value := -∞
        for each child of node do
            value := max(value, minimax(child, depth – 1, FALSE))
        return value
    else
        value := +∞
        for each child of node do
            value := min(value, minimax(child, depth – 1, TRUE))
        return value
```

Program 2. *The minimax algorithm in pseudocode*

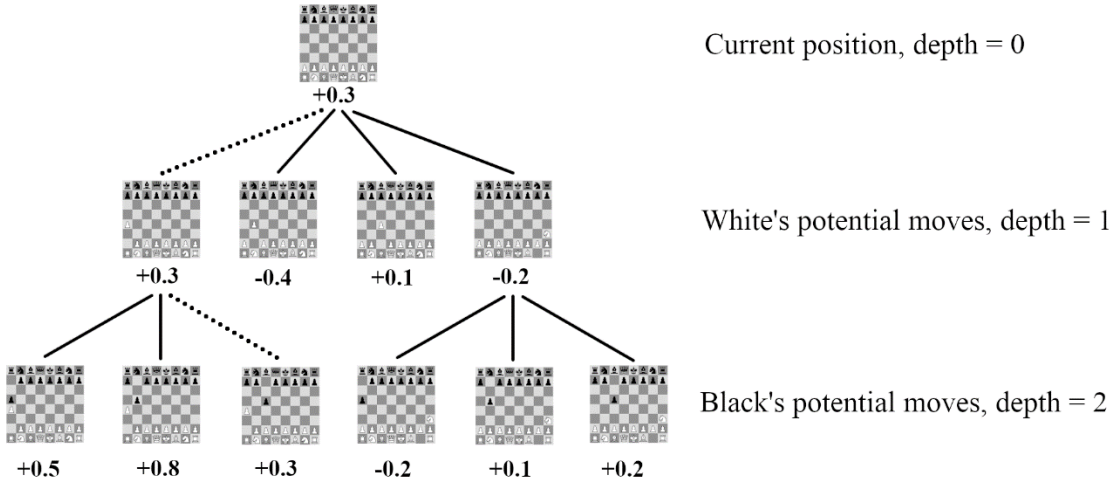
In the code above each node represents a certain chess board position. Consequently, the phrase "children of node" is then analogous to all the possible boards that result after making a move from that specific position. When the minimax function is initially called, the parameters that it requires are the current board position, the depth to which we wish to calculate (how many moves ahead to look) and a boolean called `maximizingPlayer` which specifies whose turn it currently is. In theory, we would want to set the depth to be as large as possible, but in practice it is limited by the performance of the hardware that the program is running on. The greater the depth, the more nodes have to be traversed and evaluated leading to a longer waiting time for the program to finish.

The first thing that the function does is check if the depth parameter is zero. If so, then it means that the algorithm has reached the deepest level and it can now return back up the tree with along with its calculated evaluation. However the function does not always have to reach this maximum depth, as sometimes it is possible that a certain node turns out to be a terminal leaf node. This situation can arise in a position where the game cannot continue, so when either player delivers a checkmate or if it's a draw. Naturally there are no more moves that can be generated from such a position, so as before in the depth-is-zero case we mark the winning side and return back up the tree with the result.

If on the other hand the node is not a terminal node, then the function branches into two different paths, depending on which player's turn it is to generate a move. Both of the branches essentially do the same calculations, but it's just that the maximization/minimization of the score is done for different sides. So for example when generating the moves for the white player, then the position with the highest evaluation is selected and returned. There is also a helper variable called `value` which is used to keep track of the current highest evaluation, while still searching for the best one. It has to be initialized to the lowest possible number to make sure that when we do start comparing it to the evaluations, we get the correct result. In the minimizing branch the behavior is flipped, so `value` is set to the highest possible integer and we search for the lowest evaluation score from the possible nodes.

Next, we loop through each of the child nodes calling the minimax function recursively for each one while keeping track of the best move with the `value` variable. The minimax call parameters are now as follows: the current board position is the child node's position that was just generated, the depth integer value is passed on, but one is subtracted from it in order to ensure that the base case (depth is equal to zero) is eventually reached, instead of the function getting stuck calling itself in an infinite loop. Finally the `maximizingPlayer` boolean is changed to be the opposite of what it previously was so that in the next function call the other if/else branch is chosen to alternate the turns between the two sides.

The picture on the next page attempts to visualize how the minimax algorithm chooses the move it considers best from a certain position. Note that the presented tree is very contrived with made-up evaluation scores, intended just to serve as a simple example. A real minimax tree would have considerably more nodes and a greater depth with different scores.



Picture 3. *An example of a tree generated by the minimax algorithm*

In this example, the minimax algorithm chooses a move for the white pieces. Each position has the evaluation value that is associated with it written below it. At the top of the graph is the node that represents the current state of the chess board. The row below that has all the legal moves that white can make from that position. Once the algorithm finishes, it is one of those moves that will be selected as the computer's move. The bottom row then has all the possible boards that were generated from each of the second row's nodes by moving a black piece. The lines connecting the nodes indicate the sequence of moves that took place to get to that specific position.

The algorithm works in such a way that the evaluation score is calculated only at the leaf nodes and those values are then propagated up the tree, where for each parent node the highest/lowest value child node (depending if it is the maximizing or minimizing stage) is selected. This then results in a path to the root node, that is used to pick the best move. For instance in this example the path is highlighted by the dotted lines. It starts in the bottom left branch, where the node with value of +0.3 is chosen out of the three possibilities, since it is black's move and the aim is to minimize the score by selecting the lowest one. The next step takes place in the row above which is the maximizing stage, so there the highest score is picked where +0.3 is now the highest out of the four nodes. In the end the move that is chosen is the left-most one from the second row.

4.3 Board evaluation function

So how exactly are the evaluation scores calculated? Once again, there are many different ways to do this but the ultimate goal is to obtain a value that accurately describes the power balance between the two sides on the board, while also making sure it doesn't take too long to perform these calculations.

The first thing to consider are the material imbalances in the position. If one player has more pieces on the board than their opponent, then they clearly have an advantage. One way to incorporate this into the board evaluation function is to simply count the number of pieces. Since different pieces are worth different amounts, they also need to be multiplied by their relative values. The table below shows the most common piece relative values that have been derived from empirical evidence. [14]

Table 2. *Relative values of the chess pieces*

Piece	*pawn*	*knight*	*bishop*	*rook*	*queen*	*king*
Value	1	3	3	5	9	∞

The king usually isn't given a separate value, since losing the king means losing the game. The evaluation function multiplies each white piece by its relative value and then subtracts each black pieces value from that number. When entering these values into the engine they are usually multiplied by 100 to provide a finer level of detail and are also slightly adjusted. For example, in this paper's engine knights are worth 320 points and bishops, being considered a bit better, are given 330 points.

A chess piece's value is also influenced by the square where it is located. A knight in the middle of the board controlling eight squares is much more powerful than a knight in the corner, which can only move to two squares at most. Similarly, a pawn that is about to promote to a queen is worth more than a pawn still standing on its starting square. In order to take this into consideration when calculating the evaluation function each piece has its own 8×8 array called a piece-square table, which specifies a value for each of the 64 squares on the board. The square where the piece is currently located is then added to the score. For instance one possible piece-square table for the white knight is presented below.

-50	-40	-30	-30	-30	-30	-40	-50
-40	-20	0	0	0	0	-20	-40
-30	0	10	15	15	10	0	-30
-30	5	15	20	20	15	5	-30
-30	0	15	20	20	15	0	-30
-30	5	10	15	15	10	5	-30
-40	-20	0	5	5	0	-20	-40
-50	-40	-30	-30	-30	-30	-40	-50

Picture 4. *The piece-square table for the white knight*

Another often used metric when calculating the evaluation score is piece mobility. It essentially means incorporating how many legal moves each piece has in a position into the function. Usually if one player has more room to maneuver it leads to a positional advantage, bringing about more chances to win material or deliver a checkmate. In contrast, the player with fewer potential moves more than often ends up feeling cramped with less ways to improve his pieces.

Pawns also play an important part when determining the nature of a position. Even though a single pawn is worth the least amount out of all the pieces, combined together their placement on the chess board form structures that can lead to decisive advantages or weaknesses. Evaluation functions often include this information by examining various features, such as how many pawns are connected with one another or checking how many pawns are blocking each other and adding it to the total score.

Finally, it must be mentioned that all of these aforementioned material and positional imbalances mean nothing if there is a possibility to force a checkmate on the board. Therefore if the board evaluation function spots a checkmate it assigns a sufficiently large numerical value, such as 10 000 (or -10 000 if it is black who is checkmating) to the position in order to ensure that it will be picked over all the other options. One thing that was not immediately obvious when writing the engine was that it is also useful to include the depth of a node into the evaluation score. This is because during testing it became clear that if there were multiple ways to deliver a checkmate, the computer would always select the node that was deepest in the tree, which lead to a situation where it would continuously shuffle the pieces around without ever actually delivering the final blow. By including the depth into the score, the engine favors nodes that lead to a checkmate quicker therefore fixing the problem.

5. EXAMINING THE RESULT

Now that it has been covered how the chess engine operates, we can examine its performance and consider possible areas for improvement. But what criteria or metrics could be used to estimate how well a chess engine performs? One way is to look at how many nodes it capable of going through in one second. This value of course depends highly on the hardware that the engine is running on and also varies from one position to another, but it can still give some clue as to how efficiently the program runs.

After running the engine through a few moves, the results indicated that the it could calculate around 40 000 nodes/second. For reference, this was done on ten-year-old laptop with a processor that was running at a clock frequency of 2.50 GHz. How does this compare to other engines? Well, the latest version of Stockfish running locally in the browser [15] could manage roughly 170 000 nodes/second on average. It was actually quite difficult to determine a singular value, because it varied wildly from as low as 50 000 nodes/second all the way up to 300 000 nodes/second at its peaks.

The first thing to do to improve the performance would be to upgrade the minimax algorithm to something more efficient. The alpha-beta pruning search algorithm is just that. It returns the same result as minimax, but instead of covering all the nodes it seeks to decrease that number by “pruning” away the unnecessary ones. It is slightly more complicated to implement than its predecessor, however it offers quite significant increases in performance and adding it raised the performance to 80 000 nodes/second. As a result the engine could calculate to twice the depth than was previously possible in the same amount of time. Of course it would be possible to optimize the search process even further and that is something that modern chess engines strive to do.

Though it must be said that just looking at the performance numbers does not paint the whole picture. The quality of the moves made is also highly important, however it is much more difficult to judge this feature. One method would be to let it play multiple times against other opponents (either human or computer) who have an established rating, which would then tell us how well it does. This process would be quite time consuming, so instead we’ll simply present some general observations of the engine’s level of skill made by the author.

After playing a series of games against the chess engine, it could be said that it performs relatively well. It doesn't really make simple one-move mistakes, but on the other hand it also doesn't seem to plan ahead too far either. One thing that became obvious is that the engine was quite good at punishing the opponent for his mistakes, so for example if a piece was left unguarded or a fork was available it would always take the opportunity to win material. Winning against the computer wasn't self-evident and some effort had to be made to beat it.

The engine did seem to make some questionable choices in the opening stage of the game, since at the start it often made unusual moves that were against established opening principles that are often recommended to be followed. These included moving the same piece twice and not castling soon enough. While not in any way catastrophic, this behavior could be corrected by adding a so-called opening book, which is essentially a database that contains the most tried and tested moves that are then used at the beginning stages of the game.

An improved board evaluation function would probably make the engine play better, by giving a more accurate assessment of each position, but the question then is how to decide what parameters to tweak and by how much. It is not easy to see straight away how those changes affect the performance and in the worst-case they could even make the engine play worse. Development of chess engines usually involves lots of playtesting before any changes are actually added into the source code.

One final thing that comes to mind when considering ways to improve the engine would be to somehow add parallelism to it. Dividing the work into multiple threads or cores could potentially lead to a better performance, but it would most likely require lots of careful planning and thinking of how to actually implement it in a correct manner.

6. CONCLUSION

Hopefully this paper gave some insight as to how a chess engine operates and the different intricacies that come along with it. In recent years considerable advancements have been made in the field of computer chess with the use of artificial intelligence. In 2017 Google released a research paper on AlphaZero, an entirely new type of chess engine that uses neural networks instead of the more traditional search-based algorithms. [6] Starting without any prior knowledge of the game except for the rules themselves, the engine was trained by letting it play against itself over and over. This was quite revolutionary because traditional engines rely upon years of chess knowledge that has been programmed in by human beings.

Programming chess engines is a rather broad field and there will always be new things to discover, learn and improve on. Even with the arrival of the new neural-net based chess engines, it could still be argued that it is worth to study and devote time to the more traditional search-based variants. After all, decades have been spent developing and adding knowledge to them, which can also be utilized in other areas of computer science as well. It will certainly be fascinating to see what new developments the future will bring in the world of computer chess.

7. REFERENCES

[1] Wiktionary, free content dictionary. Available: https://en.wiktionary.org/wiki/computer_chess

[2] Krešimir Josić , The University of Houston Mathematics Department. Available: https://www.uh.edu/engines/epi2765.htm

[3] Chess Programming Wiki, a repository of information about programming computers to play chess. Available: https://www.chessprogramming.org/MANIAC_I

[4] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, D. Hassabis, Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm, arXiv, 5.

[5] P. Norvig, S. Russell, Artificial Intelligence Modern Approach, 2010, p. 161

[6] C++ Reference Wiki. Available: https://en.cppreference.com/w/cpp/language/enum

[7] Chess.com, internet chess server, news website and social networking website. Available: https://www.chess.com/learn-how-to-play-chess#special-rules-chess

[8] Chess Programming Wiki, a repository of information about programming computers to play chess. Available: https://www.chessprogramming.org/Bitboards

[9] The On-Line Encyclopedia of Integer Sequences. Available: https://oeis.org/A048987

[10] Roman Hartmann, Rocechess chess programming website. Available: http://www.rocechess.ch/perft.html

[11] Edward Winter, Chess Notes, a historical chess journal. Available: https://www.chesshistory.com/winter/extra/movesahead.html

[12] Chess Programming Wiki, a repository of information about programming computers to play chess. Available: https://www.chessprogramming.org/Point_Value

[13] Lichess analysis tool. Available: https://www.lichess.org/analysis

APPENDIX A: SOURCE CODE

The full source code for the chess engine is available at the following web address: https://github.com/igorp/chess

Below is the minimax function with only the maximizing part included.

```
int Engine::minimax(board_state & position, int depth, bool maximizingPlayer)
{
    board_state possible_position;

    int score = Evaluation::evaluate(position, depth);
    if (depth == 0 || game_is_over(position))
    {
        return score;
    }

    // this is maximizer's move (choose best move for white)
    if (maximizingPlayer)
    {
        Board::reset_en_passant(position, WHITE);
        score = INT_MIN;
        for (int m = 0; m < 8; m++)
        {
            for (int n = 0; n < 8; n++)
            {
                if (position.board[m][n] > 0)
                {

                    if (position.board[m][n] == WHITE_PAWN)
                    {
                        // one move forward by pawn
                        if (position.board[m - 1][n] == 0)
                        {
                            // if not move to last rank
                            if (m - 1 != 0)
                            {
                                possible_position = Board::copy_position(position);
                                possible_position.board[m - 1][n] = WHITE_PAWN;
                                possible_position.board[m][n] = 0;

                                if (!Board::king_is_in_check(possible_position.board, WHITE))
                                {
                                    score = max(score, minimax(possible_position, depth - 1, false));
                                }
                            }
                            // if moves last rank, then promote to all pieces
                            else
                            {
                                // variable i is possible pieces from white knight to queen
                                for (char i = 2; i <= 5; i++)
                                {
                                    possible_position = Board::copy_position(position);
                                    possible_position.board[m - 1][n] = i;
                                    possible_position.board[m][n] = 0;

                                    if (!Board::king_is_in_check(possible_position.board, WHITE))
                                    {
                                        score = max(score, minimax(possible_position, depth - 1, false));
                                    }
                                }
                            }
                        }
                        // two moves from starting position by pawn
                        if (m == 6
                            && position.board[m - 1][n] == 0
                            && position.board[m - 2][n] == 0)
                        {
                            possible_position = Board::copy_position(position);
```

```
    possible_position.board[m - 2][n] = WHITE_PAWN;
    possible_position.board[m][n] = 0;

    possible_position.pawn_two_squares_white[n] = true;

    if (!Board::king_is_in_check(possible_position.board, WHITE))
    {
        score = max(score, minimax(possible_position, depth - 1, false));
    }
}
// capture to the left
if (n != 0
    && position.board[m - 1][n - 1] < 0)
{
    // not to last rank
    if (m - 1 != 0)
    {
        possible_position = Board::copy_position(position);
        possible_position.board[m - 1][n - 1] = WHITE_PAWN;
        possible_position.board[m][n] = 0;
        if (!Board::king_is_in_check(possible_position.board, WHITE))
        {
            score = max(score, minimax(possible_position, depth - 1, false));
        }
    }
    else
    {
        // variable i is possible pieces from white knight to queen
        for (char i = 2; i <= 5; i++)
        {
            possible_position = Board::copy_position(position);
            possible_position.board[m - 1][n - 1] = i;
            possible_position.board[m][n] = 0;

            if (!Board::king_is_in_check(possible_position.board, WHITE))
            {
                score = max(score, minimax(possible_position, depth - 1, false));
            }
        }
    }
}
// capture to the right
if (n != 7
    && position.board[m - 1][n + 1] < 0)
{
    // not last rank
    if (m - 1 != 0)
    {
        possible_position = Board::copy_position(position);
        possible_position.board[m - 1][n + 1] = WHITE_PAWN;
        possible_position.board[m][n] = 0;
        if (!Board::king_is_in_check(possible_position.board, WHITE))
        {
            score = max(score, minimax(possible_position, depth - 1, false));
        }
    }
    // last rank (promote)
    else
    {
        // variable i is possible pieces from white knight to queen
        for (char i = 2; i <= 5; i++)
        {
            possible_position = Board::copy_position(position);
            possible_position.board[m - 1][n + 1] = i;
            possible_position.board[m][n] = 0;

            if (!Board::king_is_in_check(possible_position.board, WHITE))
            {
                score = max(score, minimax(possible_position, depth - 1, false));
            }
        }
    }
}
```

```
    // en passant to the right
    if (n != 7 && position.board[m][n + 1] == -1
        && position.pawn_two_squares_black[n + 1])
    {
        possible_position = Board::copy_position(position);
        possible_position.board[m - 1][n + 1] = WHITE_PAWN;
        possible_position.board[m][n] = 0;
        possible_position.board[m][n + 1] = 0;
        if (!Board::king_is_in_check(possible_position.board, WHITE))
        {
            score = max(score, minimax(possible_position, depth - 1, false));
        }
    }

    // en passant to the left
    if (n != 0 && position.board[m][n - 1] == -1
        && position.pawn_two_squares_black[n - 1])
    {
        possible_position = Board::copy_position(position);
        possible_position.board[m - 1][n - 1] = WHITE_PAWN;
        possible_position.board[m][n] = 0;
        possible_position.board[m][n - 1] = 0;
        if (!Board::king_is_in_check(possible_position.board, WHITE))
        {
            score = max(score, minimax(possible_position, depth - 1, false));
        }
    }
}

if (position.board[m][n] == WHITE_KNIGHT)
{
    for (int k = 0; k < 8; k++)
    {
        int k_i = m + knight_move[k][0];
        int k_j = n + knight_move[k][1];
        if (k_i >= 0 && k_i < 8 && k_j >= 0 && k_j < 8)
        {
            if (Board::under_knight_control(m, n, k_i, k_j)
                && position.board[k_i][k_j] <= 0)
            {
                possible_position = Board::copy_position(position);
                possible_position.board[k_i][k_j] = WHITE_KNIGHT;
                possible_position.board[m][n] = 0;
                if (!Board::king_is_in_check(possible_position.board, WHITE))
                {
                    score = max(score, minimax(possible_position, depth - 1, false));
                }
            }
        }
    }
}

if (position.board[m][n] == WHITE_BISHOP)
{
    for (int k = 0; k < 4; k++)
    {
        int path_i = m + bishop_direction[k][0];
        int path_j = n + bishop_direction[k][1];
        while (path_i >= 0 && path_i < 8 && path_j >= 0 && path_j < 8)
        {
            if (Board::under_bishop_control(position.board, m, n, path_i, path_j)
                && position.board[path_i][path_j] <= 0)
            {
                possible_position = Board::copy_position(position);
                possible_position.board[path_i][path_j] = WHITE_BISHOP;
                possible_position.board[m][n] = 0;
                if (!Board::king_is_in_check(possible_position.board, WHITE))
                {
                    score = max(score, minimax(possible_position, depth - 1, false));
                }
            }
            // if next square is not empty, we stop
            if (position.board[path_i][path_j] != 0)
            {
```

```
                break;
            }
            path_i += bishop_direction[k][0];
            path_j += bishop_direction[k][1];
        }
    }
}

if (position.board[m][n] == WHITE_ROOK)
{
    for (int k = 0; k < 4; k++)
    {
        int path_i = m + rook_direction[k][0];
        int path_j = n + rook_direction[k][1];
        while (path_i >= 0 && path_i < 8 && path_j >= 0 && path_j < 8)
        {
            if (Board::under_rook_control(position.board, m, n, path_i, path_j)
                && position.board[path_i][path_j] <= 0)
            {
                possible_position = Board::copy_position(position);
                possible_position.board[path_i][path_j] = WHITE_ROOK;
                possible_position.board[m][n] = 0;
                if (m == 7 && n == 0)
                {
                    possible_position.can_castle_white[0] = false;
                }
                if (m == 7 && n == 7)
                {
                    possible_position.can_castle_white[1] = false;
                }
                if (!Board::king_is_in_check(possible_position.board, WHITE))
                {
                    score = max(score, minimax(possible_position, depth - 1, false));
                }
            }
            // if next square is not empty, we stop
            if (position.board[path_i][path_j] != 0)
            {
                break;
            }
            path_i += rook_direction[k][0];
            path_j += rook_direction[k][1];
        }
    }
}

if (position.board[m][n] == WHITE_QUEEN)
{
    for (int k = 0; k < 8; k++)
    {
        int path_i = m + every_direction[k][0];
        int path_j = n + every_direction[k][1];
        while (path_i >= 0 && path_i < 8 && path_j >= 0 && path_j < 8)
        {
            if (Board::under_queen_control(position.board, m, n, path_i, path_j)
                && position.board[path_i][path_j] <= 0)
            {
                possible_position = Board::copy_position(position);
                possible_position.board[path_i][path_j] = WHITE_QUEEN;
                possible_position.board[m][n] = 0;
                if (!Board::king_is_in_check(possible_position.board, WHITE))
                {
                    score = max(score, minimax(possible_position, depth - 1, false));
                }
            }
            // if next square is not empty, we stop
            if (position.board[path_i][path_j] != 0)
            {
                break;
            }
            path_i += every_direction[k][0];
            path_j += every_direction[k][1];
        }
    }
```

```
                }

                if (position.board[m][n] == WHITE_KING)
                {
                    for (int k = 0; k < 8; k++)
                    {
                        int k_i = m + every_direction[k][0];
                        int k_j = n + every_direction[k][1];
                        if (k_i >= 0 && k_i < 8 && k_j >= 0 && k_j < 8)
                        {
                            if (Board::under_king_control(m, n, k_i, k_j)
                                && position.board[k_i][k_j] <= 0)
                            {
                                possible_position = Board::copy_position(position);
                                possible_position.board[k_i][k_j] = WHITE_KING;
                                possible_position.board[m][n] = 0;
                                possible_position.can_castle_white[0] = false;
                                possible_position.can_castle_white[1] = false;
                                if (!Board::king_is_in_check(possible_position.board, WHITE))
                                {
                                    score = max(score, minimax(possible_position, depth - 1, false));
                                }
                            }
                        }
                    }
                    // castling kingside
                    if (m == 7 && n == 4
                        && position.board[7][5] == 0 && position.board[7][6] == 0
                        && position.board[7][7] == WHITE_ROOK
                        && !Board::under_control(position.board, 7, 4, BLACK)
                        && !Board::under_control(position.board, 7, 5, BLACK)
                        && !Board::under_control(position.board, 7, 6, BLACK)
                        && position.can_castle_white[1])
                    {
                        possible_position = Board::copy_position(position);
                        possible_position.board[7][6] = WHITE_KING;
                        possible_position.board[7][4] = 0;
                        possible_position.board[7][7] = 0;
                        possible_position.board[7][5] = WHITE_ROOK;
                        possible_position.can_castle_white[0] = false;
                        possible_position.can_castle_white[1] = false;
                        if (!Board::king_is_in_check(possible_position.board, WHITE))
                        {
                            score = max(score, minimax(possible_position, depth - 1, false));
                        }
                    }
                    // castling queenside
                    if (m == 7 && n == 4
                        && position.board[7][1] == 0 && position.board[7][2] == 0
                        && position.board[7][3] == 0
                        && position.board[7][0] == WHITE_ROOK
                        && !Board::under_control(position.board, 7, 1, BLACK)
                        && !Board::under_control(position.board, 7, 2, BLACK)
                        && !Board::under_control(position.board, 7, 3, BLACK)
                        && position.can_castle_white[0])
                    {
                        possible_position = Board::copy_position(position);
                        possible_position.board[7][2] = WHITE_KING;
                        possible_position.board[7][4] = 0;
                        possible_position.board[7][0] = 0;
                        possible_position.board[7][3] = WHITE_ROOK;
                        possible_position.can_castle_white[0] = false;
                        possible_position.can_castle_white[1] = false;
                        if (!Board::king_is_in_check(possible_position.board, WHITE))
                        {
                            score = max(score, minimax(possible_position, depth - 1, false));
                        }
                    }
                }
            }
        }
    }
    return score;
}
```

```
    // this is the minimizer's move (choose best move for black)
    else
    {
        // ...
    }
}
```