Developing a Chess Engine

LAB University of Applied Sciences
Bachelor’s Degree Programme in Information and Communications Technology
2021
Roberto Marrero Rodriguez

Abstract

Author(s) Roberto Marrero Rodriguez	Publication type Thesis	Completion year 2021
	Number of pages 44	
Title of the thesis **Developing a Chess Engine**		
Degree Bachelor's Degree Programme in Information and Communications Technology		
Name, title and organisation of the thesis supervisor Ismo Jakonen		
Name, title and organisation of the client		
Abstract The aim of this project is to create an artificial intelligence that can play chess and analyse how it has improved with the different iterations of the engine. To achieve this goal different approaches to chess engines will be explored. These different algorithms will be implemented into Python with the aim of creating a low to mid-level chess engine.		
Keywords Artificial Intelligence, Chess, Neural Networks, Minimax, Python		

Contents

1 Introduction

Artificial Intelligence (AI) is a field in computer science that is developing at a fast rate, and it had become more accessible to the public. Its social and technological impact is growing exponentially. The big tech giants like Google, Facebook, Amazon, or Microsoft have been offering services and solutions based on AI. There have been apps of different kinds being developed with this technology, like videogames, financial services, or autonomous behavior in machines. (Krittanawong, 2018.)

One of the ways to explore the advancement of AI is to investigate the history of chess engines, and how they have evolved over the years. From when they beat the best chess human player for the time Garry Kasparov in 1997 to nowadays were Google DeepMind developed a neural network that played chess and in 4 hours of training beat the best chess engine known to date. Chess is a complicated game, and it is not solved as of today and it will probably not be solved soon. So developing an Artificial Intelligence that can understand and play the game at a good level is challenging. (Kumar et al, 2021.)

> *"The human mind isn't a computer; it cannot progress in an orderly fashion down a list of candidate moves and rank them by a score down to the hundredth of a pawn the way a chess machine does. Even the most disciplined human mind wanders in the heat of competition. This is both a weakness and a strength of human cognition. Sometimes these undisciplined wanderings only weaken your analysis. Other times they lead to inspiration, to beautiful or paradoxical moves that were not on your initial list of candidates."*
> – (Garry Kasparov, 2017)

In this project, we will explore the different techniques used in chess programming and how each of them helps the machine develop a better understanding of chess. The differences between the conventional chess engine approach and the new neural network approach. After that, a chess engine has been developed with the goal of achieving a low-medium level of chess and examine how the different iterations of the evaluation function and search function affect the strength or response time of the chess engine.

2 Fundamentals of Artificial Intelligence

2.1 Artificial Intelligence

“The development of full artificial intelligence could spell the end of the human race…. It would take off on its own, and re-design itself at an ever-increasing rate. Humans, who are limited by slow biological evolution, couldn't compete, and would be super-seded.”— (Stephen Hawking, 2014)

Artificial Intelligence is the branch of computer science concerned about building smart machines to perform tasks that would normally require human intelligence. The birth of artificial intelligence conversation was started by Alan Turing, who is considered the “father of computer science”. In 1950 he asked the question, “*Can machines think?*” (Turing, 1950). From there develops a test known as the “Turing Test”, where a human interrogator would try to distinguish between a computer and a human in a text response. Nowadays, this test is an important part of the history of Artificial Intelligence (AI) and is still used as a concept in the philosophy of developing this field. (IBM, 2021.)

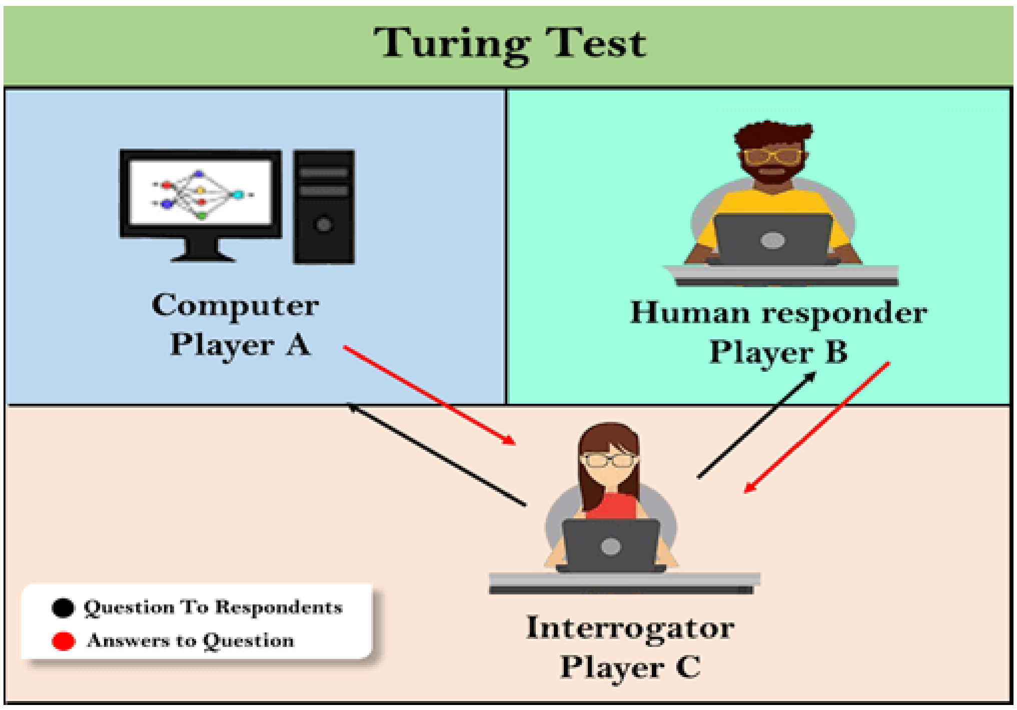


Figure 1. Turing Test. (*Turing Test in AI - Javatpoint*)

With the rapid advancement of Big Data technologies like improved computing storage and fast data processing machines, there is high interest in AI and it is evolving very fast, some of the product innovations are self-driving cars, chess or Go engines, face detection, recognition, et cetera. (Duan, Edwards, & Dwivedi, 2019.)

Stuart Russell and Peter Norvig differentiate between different types of AI.

- Systems that think like humans. They try to emulate the human thought process, like the Artificial Neural Network (ANN)
- Systems that act like humans. The study of how to get computers to perform tasks that, for the moment, humans do better than them, for example, robotics.
- Systems that think rationally. They try to imitate, logically, the rational thought of the human being, for example, expert systems. The study of the calculations that make it possible to perceive, reason and act.
- Systems that act rationally. ideally, they are those who try to imitate rational human behaviour, such as intelligent agents.

(Russell & Norvig, 2002.)

This is one of the many ways to classify the different types of AI. Another way to classify them would be weak AI and strong AI. Where the weak AI is trained and focused to perform specific tasks, it is also known as Narrow AI. We are surrounded by this kind of AI since it is the most popular kind, the most known being Apple’s Siri and Amazon’s Alexa. On the other hand, we have “strong AI”, which is made up of Artificial General Intelligence (AGI) and Artificial Super Intelligence (ASI). AGI and ASI are theoretical forms of AI where the machine would have an intelligence equal or superior, respectively, to humans. There are no practical examples in use today, but it is still developing. Fujitsu-built K, one of the fastest supercomputers, is one attempt at achieving strong AI, but considering it took 40 minutes to simulate a single second of neural activity, it is difficult to determine whether strong AI will be achieved in our foreseeable future. As image and facial recognition technology advances, it is likely we will see an improvement in the ability of machines to learn and see. For now, the best examples of such technology will only be found in science fiction. (IBM, 2021.)

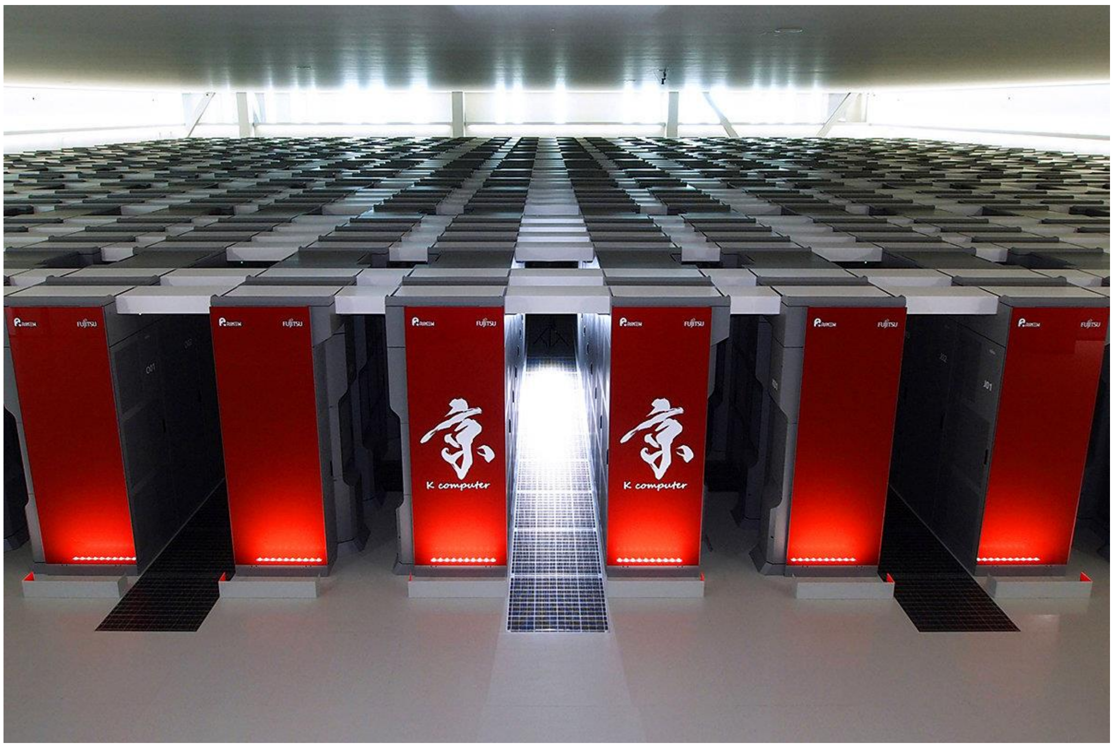


Figure 2. Fujitsu-built K supercomputer. (Hornyak, 2013)

2.2 Machine Learning & Deep Learning

Both machine learning and Deep Learning are sub-field to artificial intelligence, but deep learning is also a sub-field of machine learning. Machine learning is the application of Artificial Intelligence that provides systems to learn and improve from experience without being explicitly programmed. Machine learning focuses on developing computer programs that can have access to data and can use it and analyze it to learn for themselves. (Raschka & Mirjalili, 2017.)

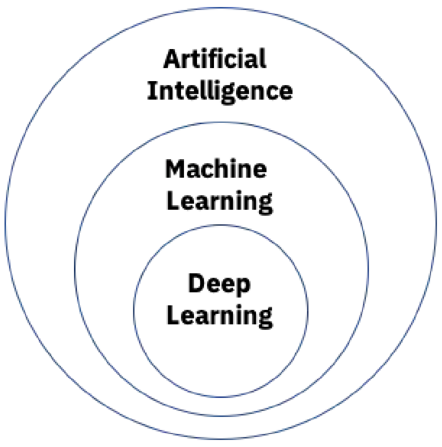


Figure 3. Artificial Intelligence Diagram(IBM, 2021)

This process begins with observations of data and trying to look for patterns in the data and change the behaviour accordingly. The principal objective is to allow the computer to learn automatically without human intervention or assistance. Otherwise, we have deep learning which is comprised of artificial neural networks. The deep stands to refer to a neural network comprised of more than three layers. In Figure 4 there is the general representation of the deep neural network.

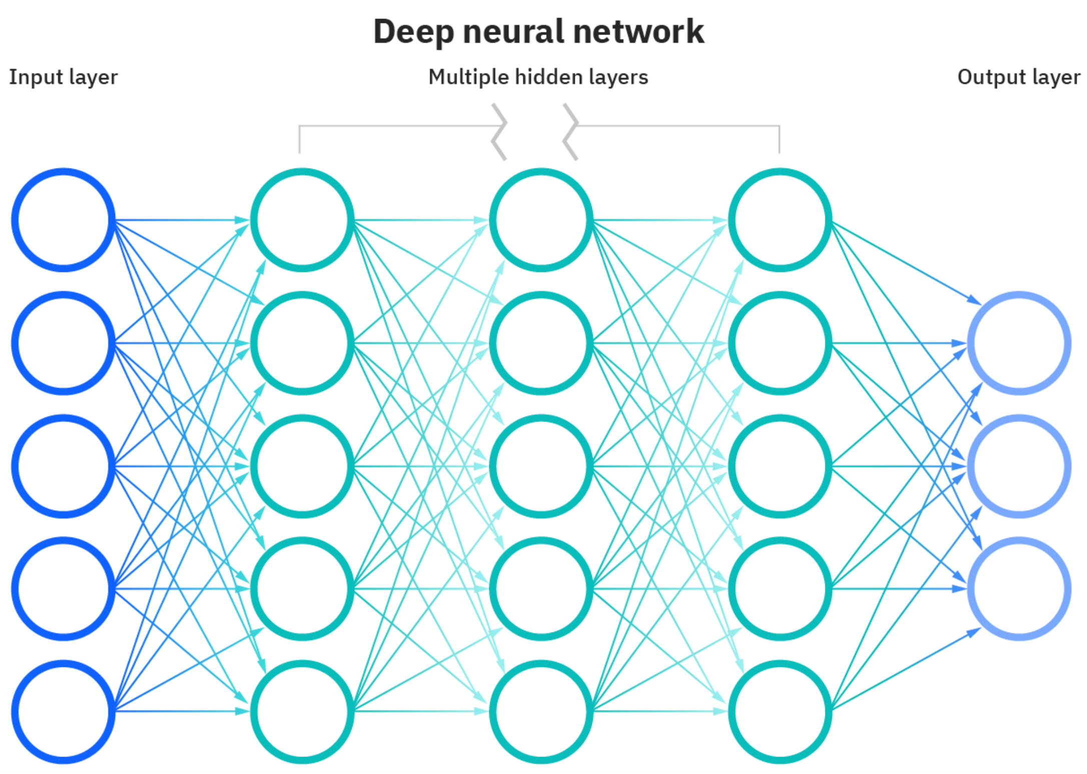


Figure 4. Diagram of a deep neural network(IBM, 2021)

To understand neural networks an example will be used. In this case, is the recognition of handwritten digits. For us humans, we have no trouble recognizing Figure 5 as 504192. It seems like a simple task for us, but how we arrive at the conclusion is not trivial. Humans have a visual cortex that consists of 140 million neurons with tens of billions of connections between them, not only that, but we have also more than one visual cortex doing very complex image processing to understand what we are seeing. Recognizing handwritten digits is not easy, rather humans are extremely good at understanding what our eyes show us. (Nielsen, 2015.)

Figure 5. Handwritten numbers (Nielsen, 2015)

This problem arises when you try to write a computer program that tries to recognize the digits from Figure 5. What humans do effortlessly suddenly become very difficult. Some of the intuition that we may have like, an eight has 2 loops, one at the top and another one at the bottom, or a nine has one loop at the top and one stroke at the right, are not simple to express algorithmically. Making rules that precise will lead to a lot of exceptions, caveats, and special cases. Artificial Neural Networks approach this problem in a different way. The main focus is to have a large number of handwritten digits known as training examples. Then, making the system learn from these training examples. So basically, instead of programming the different algorithms on what defines an 8 or what defines a 9, we will let the program infer its own rules. (Nielsen, 2015.)

The neurons, on a computer science basis, are a node that contains a value between 0 and 1. For this example, every number that we feed to the neural network will be on a 28x28 grid of pixels, creating a total of 784 pixels. The number inside the neurons is called activation and will represent the gray-scale value of the corresponding pixel, ranging from zero for black pixels up to one for white pixels. The first layer of this network will be composed of 784 neurons each connected to a pixel in the input image. Eventually, the last layer of the neural network will be 10 neurons containing the 10 different outputs that we are looking for, from number 0 to number 9. The activation of the neuron on this output layer holds means how much does the system thinks the given image corresponds to a given digit. Then, we have the hidden layers in between the previous 2 layers. The number of hidden layers can be arbitrary, for this example we will use 2 hidden layers with 16 neurons each. This is represented in figure 6. (3Blue1Brown, 2017.)

Essentially, the neural networks work with the activations on one layer determining the activations of the next layer. How the connection between layers is achieved is the heart of the network and is meant to be similar to how some group biological neurons firing cause certain others to fire.

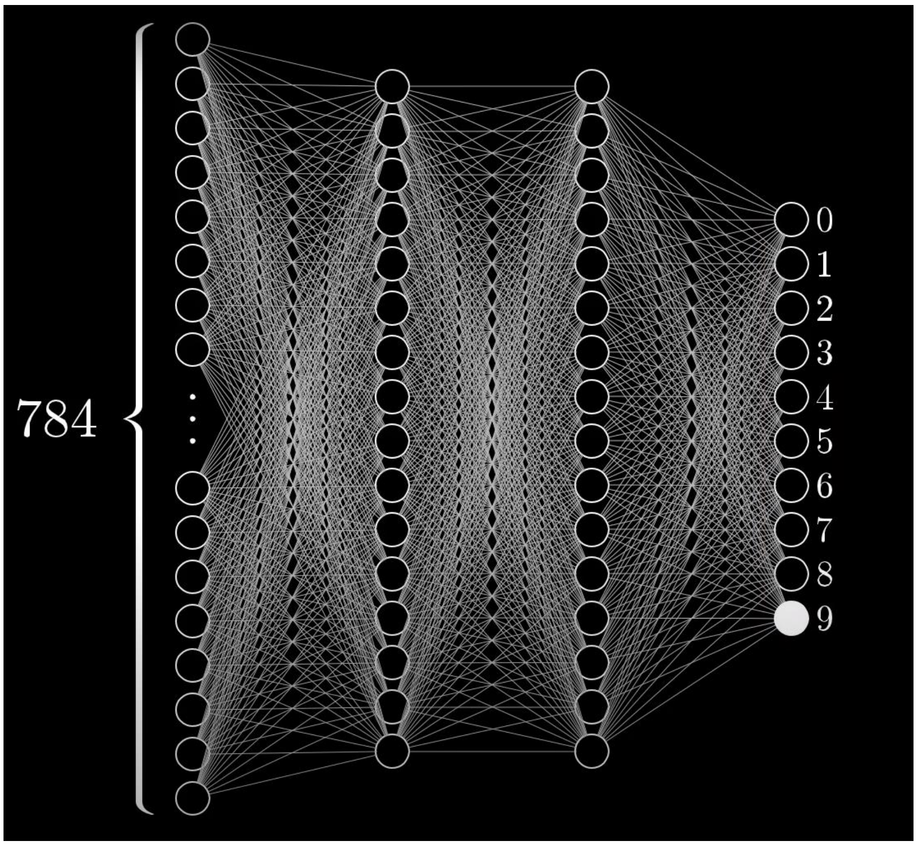


Figure 6. Neural network example (3Blue1Brown, 2017)

The math behind how one layer influences the next is, each connection will have a weight assigned between the neuron from one layer to the next, these weights are just numbers. Then we will take all the activations from one layer and compute their weighted sum according to these weights. This will give a any number, but the neurons can only store values from zero to one, so some function that squishes the value of the real number into the range between zero and one. There are several functions that do this kind of work, those being the sigmoid function or the Rectified Linear Unit (ReLU) function. One of the most popular nowadays is the ReLU. To add to this new function, we need to add some bias for inactivity. This will help when for example, you want the neuron to light up when the weighted sum is bigger than 10 instead of bigger than 0. This bias will be added before using the ReLU function. So, the weight will tell you what pixel pattern this neuron in the second layer is picking up on and the bias tell you how high the weighted sum needs to be before the neuron start getting meaningfully active. This is only one neuron every other neuron will be connected to all neurons form the previous layer and each of those connections will has its own weight associated with it, and each one has some bias. In the neural network presented in Figure 6 on the first hidden layer of 16 neurons that is a total of 784 times 16 weight and 16 biases. This is only the connection from the first layer to the second. The connection between other layers also has weights and biases associated with them. This network has 13.002 total weights and biases. So, what the computer has to learn is, how to tweak all of these weights and biases to achieve the wanted result. (3Blue1Brown, 2017.)

3 Development of Computational Chess

3.1 History

In the early years of chess engines, the idea of having a computer that played chess at the same level as the best grandmasters was unimaginable. Since chess requires a lot of creativity and has millions of possibilities, the average number of legal different positions that you can find is around 10^40 and 10^50(Holcomb). Nevertheless, nowadays it has been proven that chess engines are far superior to the best grandmasters.

The first instance of a chess engine defeating a world champion happened in 1997, where Deep Blue, an engine created by IBM, defeated Garry Kasparov. This happened over 20 years ago and today the advances in Artificial Intelligence and computer hardware are huge. (“IBM100 - Deep Blue,” 2012.)

Figure 7. Garry Kasparov playing vs Deep Blue 1997 (George Widman)

It should be noted that chess engines and humans approach chess in a different way. Fundamentally they both analyze and calculate moves and search ahead to predict how the game would go if they followed that line. The major difference is, when the best chess player Kasparov can only analyze 3 to 5 positions per second, Deep Blue was analyzing at the time 200 million positions per second (*IBM100 - Deep Blue*, 2012). Even with this disparity, the games ended up even. Out of the 6 games played between them, Kasparov won 1 tied 3 and lost 2. We can make the conclusion that humans are much more efficient at choosing the correct line to follow. (“IBM100 - Deep Blue,” 2012.)

The major fight of chess engines is knowing which lines are better to pursue, and it has proven that this is quite an arduous task. Having the computational capacity of an engine and the human intuition would be perfect, but how do humans get that game sense and how can we recreate it on a chess engine? Well, this concept is very abstract, the best chess players play, analyze and observe millions of games which gives them that game sense. As for today, this intuition has not yet been programmed fully. (Podlesak, 2019.)

There have been different approaches to this, the main ones being Stockfish, which is the strongest traditional chess engine to date, having an ELO rating of ~3550, to put this in perspective the best chess player in the world Magnus Carlsen has an ELO of ~2862 and his best is 2882. (FIDE Ratings.)

On the other hand, there has been a fresh approach to chess engines, one that tries to replicate the learning experience of humans to make it more efficient. This is achieved by using machine learning instead of hard coding what makes a good position as stockfish does. With machine learning, the computer only gets the game of chess and basic rules and learns by playing games against themselves. This has been a success and the most famous engine that uses this approach is AphaZero which is developed by DeepMind a business owned by Google that focuses on Artificial intelligence. (Podlesak, 2019.)

Figure 8. AlphaZero vs Stockfish historical match (Pete, 2019)

Lastly, the stockfish team in 2020 joined the traditional chess engine with neural networks. They created what is today as Stockfish NNUE (Efficiently Updatable Neural Networks). On September 02, 2020, Stockfish 12 was released with a huge jump in playing strength due to the introduction of this technology. The architecture of Stockfish NNUE is extremely different from the neural networks AlphaZero uses. AlphaZero uses an extremely deep, convolutional neural network with as many as 40 layers.

3.2 Traditional Chess engines

Traditional chess engines use complex evaluation functions and intelligent search algorithms to find the best possible move. Their power is also related to how much CPU processing power the phone, computer, or server has. The more powerful and plentiful the CPUs, the stronger the engine becomes. (Team (CHESScom)) One of the modules of the engine is the movement generator, which is a basic part of a chess engine and its implementation depends heavily on the board representation. There are two types of move generation. In the pseudo-legal move generation, pieces will obey normal rules of movement, but the move is not checked beforehand to see if, for example, it will leave the king in check. The legal move generation will create only legal moves. This will make this process take longer since checking if the king is not going to be left in check after the move. When a piece is defending the king from another enemy piece and it cannot move is called a pin. Pins create the most difficulty for the legal move generator. (Schaeffer, Powell, & Jonkman, 1983.)

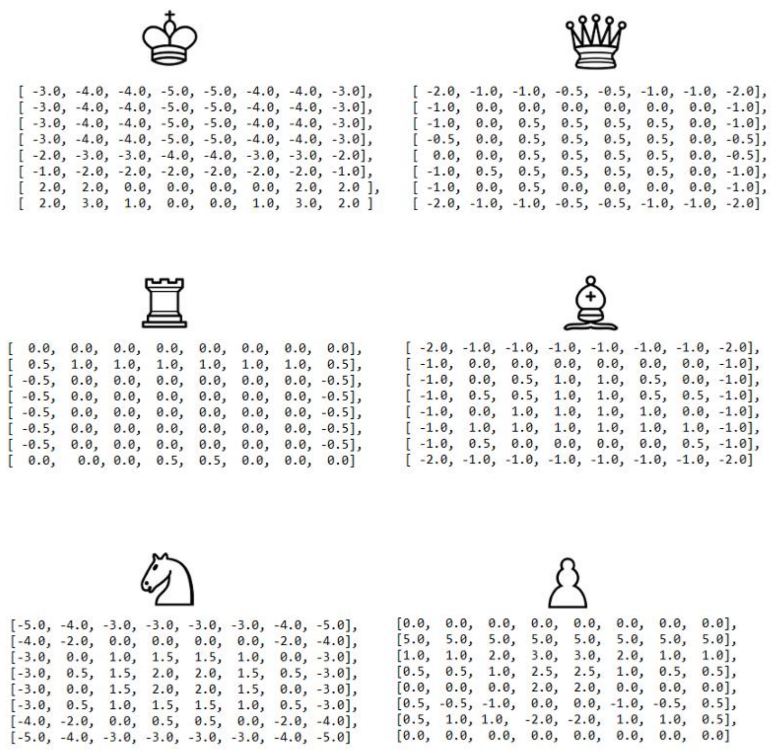
3.2.1 Evaluation function

The evaluation function is the most important module, nowadays the main improvement in chess engines comes in changing the evaluation function. The input for this module is a chess position, and the output is a number. If this number gives us an evaluation of 0, it means the position is equal for both players. The higher the positive number more advantage for white, the lower the negative number, the more advantage for black. There is different data the algorithm looks to come out with an evaluation. This has been hardcoded to mimic a grand mastermind. This is fundamental for understanding chess and chess engines. Stockfish uses the following methods to calculate the evaluation: (CPW-Evaluation.)

- Material

A value is assigned to each piece in chess, for example, a knight is worth 3 points and a queen 9. And the material will be the sum of all available pieces. Material value is what influences most of the evaluation. (Material - Chessprogramming Wiki.)

- Piece-Square Tables



```
[ -3.0, -4.0, -4.0, -5.0, -5.0, -4.0, -4.0, -3.0],
[ -3.0, -4.0, -4.0, -5.0, -5.0, -4.0, -4.0, -3.0],
[ -3.0, -4.0, -4.0, -5.0, -5.0, -4.0, -4.0, -3.0],
[ -3.0, -4.0, -4.0, -5.0, -5.0, -4.0, -4.0, -3.0],
[ -2.0, -3.0, -3.0, -4.0, -4.0, -3.0, -3.0, -2.0],
[ -1.0, -2.0, -2.0, -2.0, -2.0, -2.0, -2.0, -1.0],
[  2.0,  2.0,  0.0,  0.0,  0.0,  0.0,  2.0,  2.0 ],
[  2.0,  3.0,  1.0,  0.0,  0.0,  1.0,  3.0,  2.0 ]
```

```
[ -2.0, -1.0, -1.0, -0.5, -0.5, -1.0, -1.0, -2.0],
[ -1.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0, -1.0],
[ -1.0,  0.0,  0.5,  0.5,  0.5,  0.5,  0.0, -1.0],
[ -0.5,  0.0,  0.5,  0.5,  0.5,  0.5,  0.0, -0.5],
[  0.0,  0.0,  0.5,  0.5,  0.5,  0.5,  0.0, -0.5],
[ -1.0,  0.5,  0.5,  0.5,  0.5,  0.5,  0.0, -1.0],
[ -1.0,  0.0,  0.5,  0.0,  0.0,  0.0,  0.0, -1.0],
[ -2.0, -1.0, -1.0, -0.5, -0.5, -1.0, -1.0, -2.0]
```

```
[  0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0],
[  0.5,  1.0,  1.0,  1.0,  1.0,  1.0,  1.0,  0.5],
[ -0.5,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0, -0.5],
[ -0.5,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0, -0.5],
[ -0.5,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0, -0.5],
[ -0.5,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0, -0.5],
[ -0.5,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0, -0.5],
[  0.0,   0.0, 0.0,  0.5,  0.5,  0.0,  0.0,  0.0]
```

```
[ -2.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -2.0],
[ -1.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0, -1.0],
[ -1.0,  0.0,  0.5,  1.0,  1.0,  0.5,  0.0, -1.0],
[ -1.0,  0.5,  0.5,  1.0,  1.0,  0.5,  0.5, -1.0],
[ -1.0,  0.0,  1.0,  1.0,  1.0,  1.0,  0.0, -1.0],
[ -1.0,  1.0,  1.0,  1.0,  1.0,  1.0,  1.0, -1.0],
[ -1.0,  0.5,  0.0,  0.0,  0.0,  0.0,  0.5, -1.0],
[ -2.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -2.0]
```

```
[-5.0, -4.0, -3.0, -3.0, -3.0, -3.0, -4.0, -5.0],
[-4.0, -2.0,  0.0,  0.0,  0.0,  0.0, -2.0, -4.0],
[-3.0,  0.0,  1.0,  1.5,  1.5,  1.0,  0.0, -3.0],
[-3.0,  0.5,  1.5,  2.0,  2.0,  1.5,  0.5, -3.0],
[-3.0,  0.0,  1.5,  2.0,  2.0,  1.5,  0.0, -3.0],
[-3.0,  0.5,  1.0,  1.5,  1.5,  1.0,  0.5, -3.0],
[-4.0, -2.0,  0.0,  0.5,  0.5,  0.0, -2.0, -4.0],
[-5.0, -4.0, -3.0, -3.0, -3.0, -3.0, -4.0, -5.0]
```

```
[0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0],
[5.0,  5.0,  5.0,  5.0,  5.0,  5.0,  5.0,  5.0],
[1.0,  1.0,  2.0,  3.0,  3.0,  2.0,  1.0,  1.0],
[0.5,  0.5,  1.0,  2.5,  2.5,  1.0,  0.5,  0.5],
[0.0,  0.0,  0.0,  2.0,  2.0,  0.0,  0.0,  0.0],
[0.5, -0.5, -1.0,  0.0,  0.0, -1.0, -0.5,  0.5],
[0.5,  1.0, 1.0,  -2.0, -2.0,  1.0,  1.0,  0.5],
[0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0]
```

Figure 9. Piece-Square Table example (Hartikka, 2017)

Each piece receives a bonus value depending on what square the piece is in. Each piece of each color has its own table. The value of a square for a certain piece can vary through the game and with this we can achieve different goals, like pawns advancing in the end-game or knights and bishop developing in the opening stages of the game. To make a semi-decent chess engine just with material and Piece-Square Tables are enough, but nowadays we will need much more to compete with top-level engines. (Piece-Square Tables - Chess-programming Wiki.)

- Pawn Structure

This term is used to describe the position of the pawns on the board. This will ignore all the other pieces. To determine a good or bad pawn structure we can look if we have doubled pawns, which means, having 2 pawns on the same file or having a pawn with a clear path for promotion. This last situation is known as a passed pawn. There are more things to take into consideration for pawn structure like isolated pawns, pawn islands, etc. This will be all hardcoded considering the benefits and inconveniences of each situation. (Pawn Structure - Chessprogramming Wiki.)

- Evaluation of Pieces

Every piece can have its evaluation changed depending on the board state, for example, if the rook is on an open file the value of the rook will go up significantly. On the other hand, if your black square bishop is trapped and cannot move because it is behind black squared pawns, the value of that bishop will go down. Each of the 6 different pieces will have some specific rule to either make its value better or worse. (Evaluation of Pieces - Chessprogramming Wiki.)

- Evaluation Patterns

There are some positions that will require additional knowledge to make a correct evaluation, one of the most famous examples will be when you fianchetto your bishop and its part of your king defensive formation it is undesirable to exchange it for a knight or bishop. (AKDEMIR.)

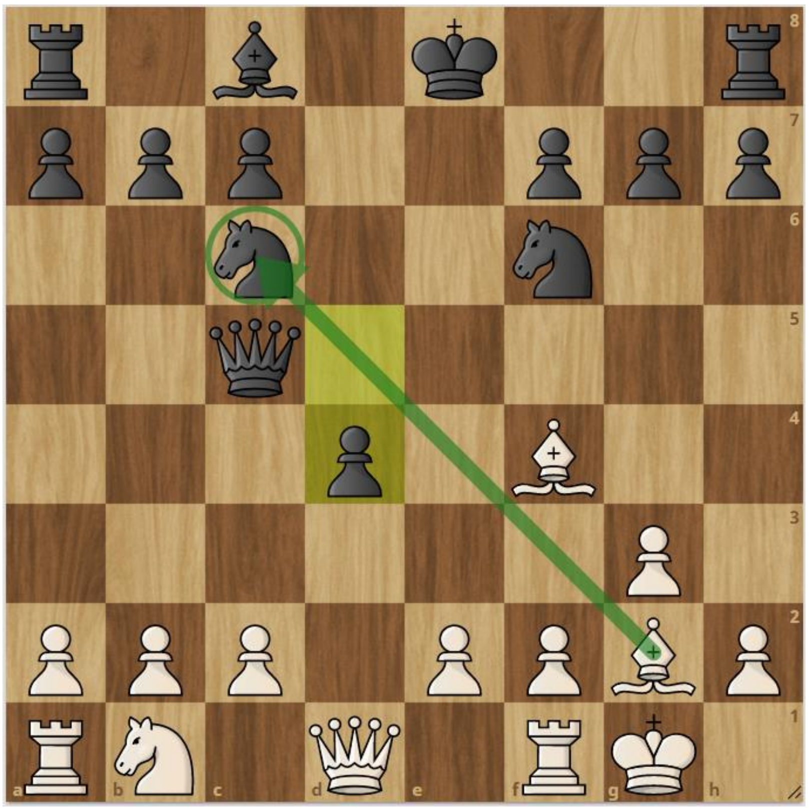


Figure 10. Example of position showing undesirable bishop-knight exchange.

- Mobility

This term refers to the number of legal moves the player has on the given position. Generally, the more moves you have, the better your position is. Since you have more piece activity. (Chess Programming Part VI.)

- Center Control

Controlling the center squares in chess is a really good strategy that generates space and allows pieces to get to the desired spaces. Normally the center is controlled by pawns, but we have seen some modern chess openings where the center is controlled by pieces from far away. (Chess Programming Part VI.)

- Connectivity

It is important that every piece is well defended. So having defended pieces will make the evaluation go in your favour while having undefended pieces will work against the evaluation function. (Levinson & Weber, 2001.)

- King Safety

For humans, king's safety is the number one priority in chess. This task is very hard to put into code. There are a lot of factors that can go inside the calculation of king's safety, for example, how well structured is the pawn shield in front of the king, how many attacking pieces does the enemy have, are there files open near the king. (Tesauro, 2001.)

- Tempo

Tempo refers to the ability to make moves. The fewer moves you need to get a piece to a certain square you will gain a tempo. This is important in chess because if you always make a threat on every move, your opponent loses a tempo responding to it, but the moment you make a passive move, your opponent can take the initiative and gain tempo. (Tempo - Chessprogramming Wiki.)

These are the variables that go into a chess evaluation function. It is to be noted that every single detail has been human conclusions and hardcoded. These concepts are a part of the experience of humans after centuries of chess games being played. Nevertheless, every time we discover new positions or different tactical approaches to different positions, this is what makes conventional chess engines flawed. It is impossible to make a truly objective evaluation function, this is the reason why the evaluation function keeps changing every year to keep up with the new chess discoveries and cover up these little details. (Chess Programming Part VI.)

3.2.2 Search function

This module will be the one in charge to calculate the different variants given a position. This is a very complex thing to do, and once again it tries to mimic humans. You may be able to make a move that improves your position but leads to forced mate in several moves from the opponent. Calculating the variables optimally and knowing which will be the best path to pursue is critical for winning at chess. There have been different algorithms that chess engines have used over the years. We can differentiate them into two types: brute-force search and selective search. In the earlier days, the selective search was favoured. This type had a major risk since it had the possibility to oversee some tactics. Nowadays, with the amount of computing power available, programs are closer to a brute-force search, but they still use some characteristics of the selective search. The most popular search algorithms used today are minimax and alpha-beta. (Search - Chessprogramming Wiki.)

- Minimax

John von Neumann, in 1928 classified chess as a two-player zero-sum game with perfect information. And, as he stated, there will be an optimal solution to it. First, to understand this algorithm the concept of a zero-sum game needs to be clear. A zero-sum game is when one person's gain is equivalent to another's loss, so the net change in benefit is zero, for example, if in a game of chess player A has an advantage of 1 queen is because player B has an advantage of -1 queen. One player can only win what another player has lost. The previous example is a very simplified version of chess, but you can get a similar conclusion after analyzing a chess position.

Now that the concept of a zero-sum game is understood, the minimax algorithm works with the idea that both players will go for the best possible move. This is achieved by doing the play that suits the opponent the less. So basically, you are minimizing the maximum loss. (Beal, 1982.)

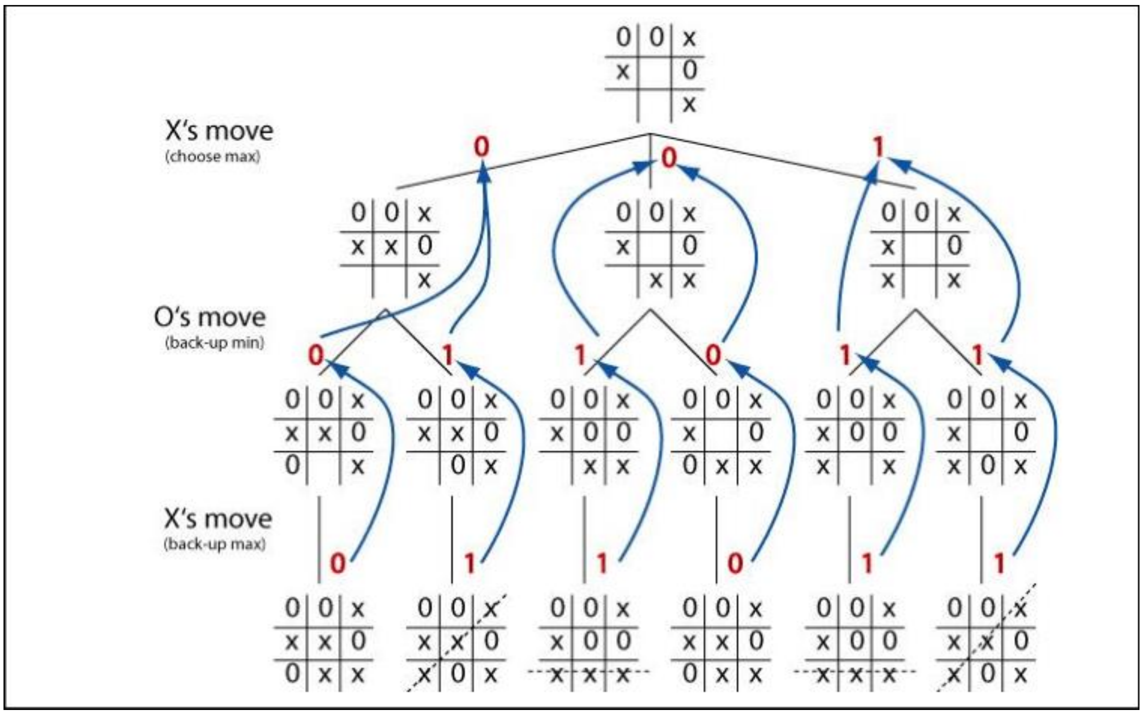


Figure 11. Example of Minimax Algorithm with Tic-Tac-Toe(Game Tree for Tic-Tac-Toe Game Using MiniMax Algorithm.)

In the previous figure, we can see a minimax algorithm in action, it is working with a depth of 3. All of X's moves will be calculated and then all of O's possible answers till we reach 3 moves for the starting position. Player X will choose the highest number and Player O will choose the lowest number. So, even if you choose the left or the middle path, you can still win as player X, if player O plays perfectly you will not win. In this example, we are just dealing with values of -1 or 1 in chess. The evaluation function will give you a higher range of numbers, but the same concept still applies. One player will be denominated max and he will choose the highest value and the other player will be min, and he will choose the smaller value.

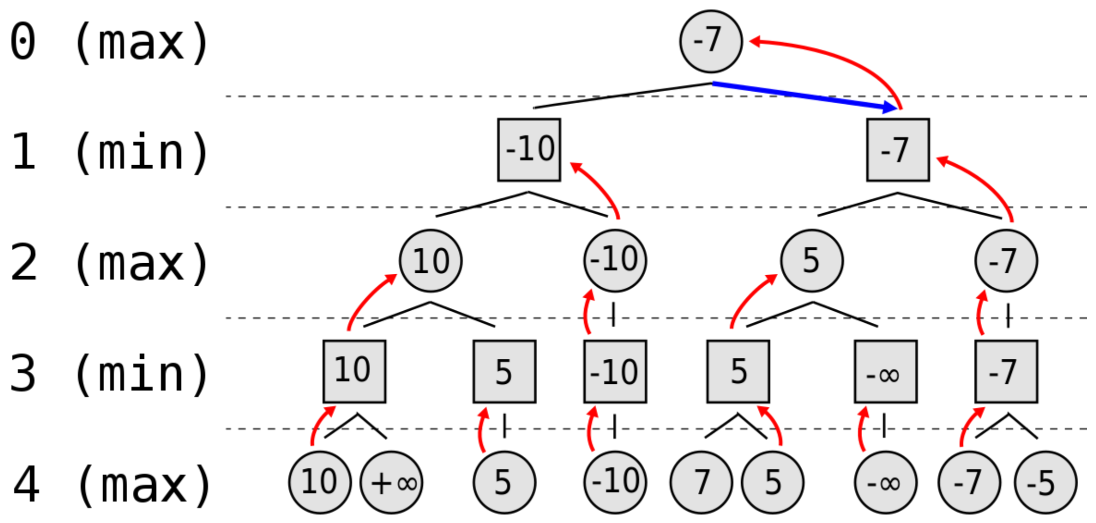


Figure 12. Example of Minimax algorithm using numeric values. (“Alpha–Beta Pruning,” 2021)

In this new example, we have a depth of 4. On move 3, min chooses the lowest value from move 4. Min chooses 10 since 10 is smaller than infinite, it also chooses -7 since -7 is lower than -5. On the other hand, on move 2 the player max chooses the highest possible number from the 3rd row and this continues till we reach row 0. One important aspect that can be observable is the number of different states and positions are derived from the first, the growth is exponential. In the middle game of a chess game on average, you have 30 possible plays. This would mean that having a depth of 5 will lead to 30^5=24.300.000 positions. Even by today's standards of technology, it will be impossible to approach high depths.

- Alpha-beta pruning

Alpha-beta algorithm is an improvement of the minimax search algorithm that reduces the number on a large scale the number of nodes evaluated. Stockfish 12 is using, with additional improvements this algorithm. To illustrate this with a real-life example, suppose somebody is playing chess, and it is their turn. Move "A" will improve the player's position. The player continues to look for moves to make sure a better one hasn't been missed. Move "B" is also a good move, but the player then realizes that it will allow the opponent to force checkmate in two moves. Thus, other outcomes from playing move B no longer need to be considered since the opponent can force a win. The maximum score that the opponent could force after move "B" is negative infinity: a loss for the player. This is less than the minimum position that was previously found; move "A" does not result in a forced loss in two moves (Alpha–Beta Pruning, 2021). To further explain how it works we will use as an example the next tree:

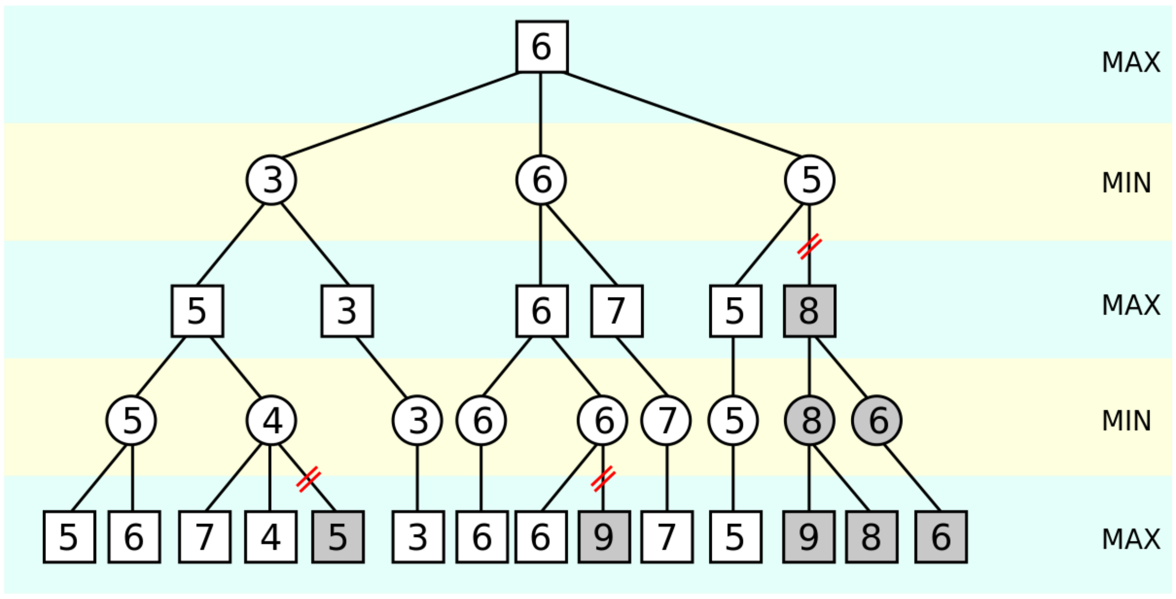


Figure 13. Alpha-beta pruning example (Alpha–Beta Pruning, 2021)

In this example, we have the max player playing with the white pieces. It has calculated the next possible results with a depth of 4. First, starting at the bottom min chooses between 5 and 6 and it chooses 5. Then min has to choose again between 7, 4, or 5, this time it stops evaluating positions at value 4 because the next move will be max turn to play and it will have to choose between 5 or 4, and since max is always going to pick 5 (the highest number possible), there is no need to keep exploring that branch since max will not pick it so we can discard it, or in the case of this algorithm, prune it. On the second main branch. We start from the bottom again. Min chooses 6 and then it has the option to choose between 6 or 9, and pruning happens again. Since we have the same value in both children nodes and max will choose the highest it serves no purpose to continue exploring that branch, so we prune it. On the third main branch, we will start at depth 2 where min has to choose between 5 or 8. When the value of 5 gets evaluated we immediately prune the rest of the tree since we now max will choose 6 or higher, there is no point to keep exploring the rest of the tree to get a number lower than 5.

To translate this concept to programming we have two limits α and β, that will correspond to the most convenient evaluation at the moment. The initial value will be -∞ and +∞ and will be updating when different variants will be evaluated. The pseudo-code a minimax alpha-beta pruning is as follows:

```
function alphabeta(node, depth, α, β, maximizingPlayer) is
    if depth = 0 or node is a terminal node then
        return the heuristic value of node
    if maximizingPlayer then
        value := −∞
        for each child of node do
            value := max(value, alphabeta(child, depth − 1, α, β,
FALSE))
            α := max(α, value)
            if α ≥ β then
                break (* β cutoff *)
        return value
    else
        value := +∞
        for each child of node do
            value := min(value, alphabeta(child, depth − 1, α, β,
TRUE))
            β := min(β, value)
            if β ≤ α then
                break (* α cutoff *)
        return value
(* Initial call *)
alphabeta (origin, depth, −∞, +∞, TRUE)
```

Figure 14. Pseudo-code of minimax with alpha-beta pruning (Alpha–Beta Pruning, 2021)

As previously mentioned, this algorithm is very similar to minimax but with a well-written program, a standard minimax tree with x nodes can be reduced close to the square root of x nodes. This is heavily reliant on how well-ordered the tree is. If the best move is always explored first, you eliminate the most nodes, but always knowing what move will be the best in a given position is a complex task. Therefore, good move ordering is extremely important, and it is where a lot of the effort in writing a successful chess engine resides. To get this result the typical move ordering will be:

1. Principal variation move (a move that was the previous iteration of an iterative deepening framework for the leftmost path)
2. Hash move from hash tables
3. Winning captures/promotions

4. Equal captures/promotions
5. Killer moves
6. Non-captures sorted by history heuristic

One of the problems that may arise is the Horizon effect. This effect is caused by the depth limitation of the search algorithm. This happens when a negative event is inevitable but postponable. The engine will only be able to analyze a partial part of the search tree, it will choose a move that will seem to avoid the threat, but this is not the case.

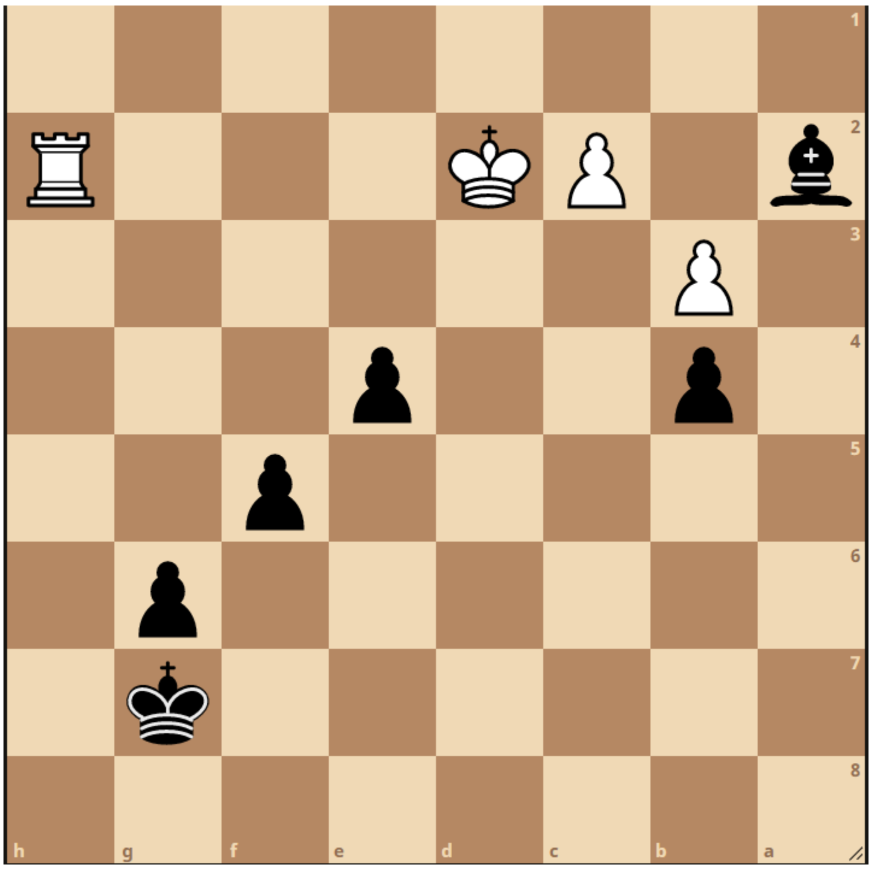


Figure 15. Example of the horizon effect

In the situation presented in figure 15, the black bishop is trapped. No matter what the black does white can always manoeuvrer the rook to a1 in two moves and capture the bishop in the third move. If we have a chess engine with a depth of 6, what could happen is that the best move for black suggested by the engine will be to push pawn e3, just to force the king to capture the pawn, then pushing the remaining pawns to force the king to keep capturing the pawns in an attempt to save the bishop, but that only will delay the capture of the bishop since it’s trapped and will lose you 3 passed pawns. Probably the best line of play for black will be to exchange the bishop for a pawn and try to hold the game with connected pass pawns against a rook. (Scoones, 2007) To combat this effect, chess engines implement the quiescence search. The main goal of the quiescence search is to not have a fixed depth for the evaluation but to have the search module analyze the variants until a stable position is reached so it can be evaluated statically. This is achieved by forcing the engine to go deeper into volatile positions where there are more captures or moves that can destabilize the evaluation function than quiet positions. A quiet position will be one that has no captures or threats. The pseudocode for the quiescence search to illustrate the concept algorithmically: (Quiescence Search - Chessprogramming Wiki.)

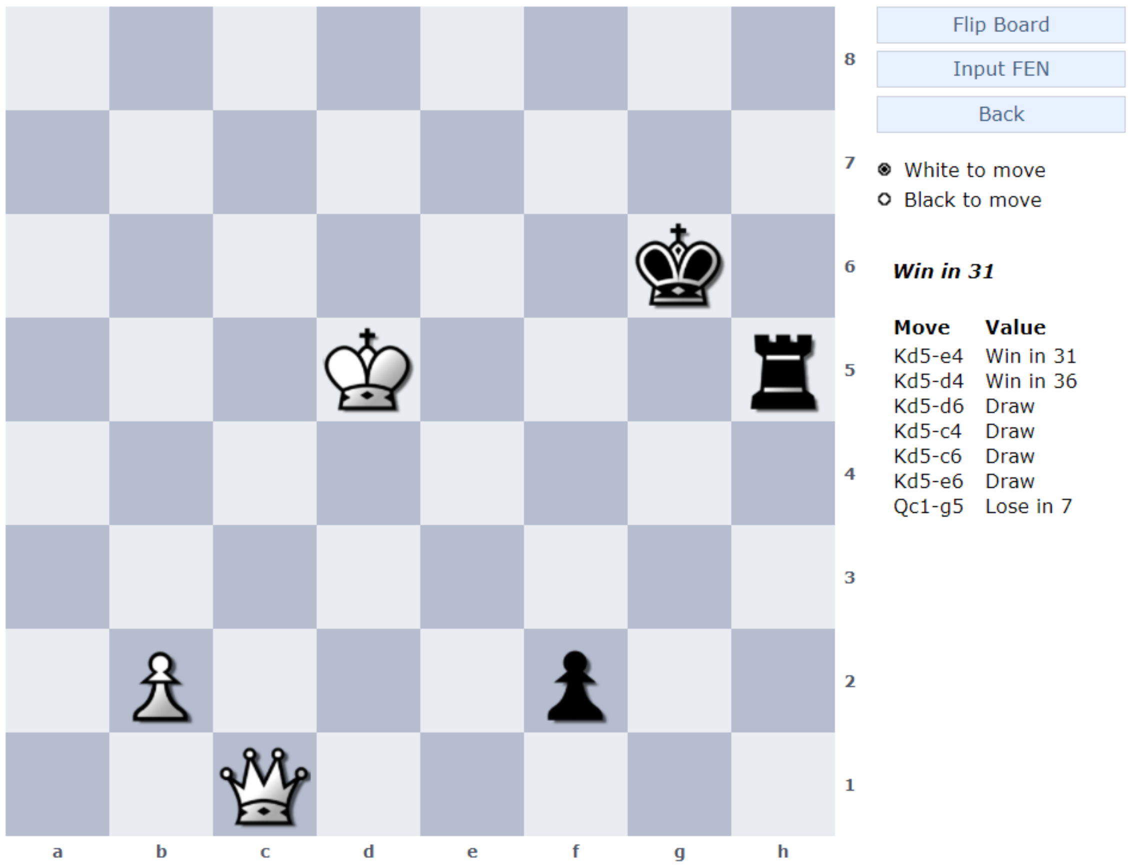
```
function quiescence_search(node, depth) is
    if node appears quiet or node is a terminal node or depth = 0
then
        return estimated value of node
    else
        (recursively search node children with quiescence_search)
        return estimated value of children

function normal_search(node, depth) is
    if node is a terminal node then
        return estimated value of node
    else if depth = 0 then
        if node appears quiet then
            return estimated value of node
        else
            return estimated value from quiescence_search(node, rea-
sonable_depth_value)
    else
        (recursively search node children with normal_search)
        return estimated value of children
```

Figure 16. Pseudocode of quiescence search(Quiescence Search, 2021).

3.2.3 Chess databases

Another important factor to consider is the use of game databases. There are millions of games stored in the different chess databases. During the first moves of the game these databases are very useful, we can look at how many times victory was achieved with certain openings. The best engines in the world still use opening books from the best chess Grandmasters. This eliminates the need for the engine to calculate the best lines during approximately the first ten moves of the game, where the positions are extremely open-ended and therefore computationally expensive to evaluate. As a result, it places the computer in a stronger position using considerably fewer resources than if it had to calculate the moves itself. (Opening Book - Chessprogramming Wiki.)

On chess engines, the endgame is approached by Nalimov Tablebases. This is a database that stores all the positions with a small piece count. These positions are already determined as winning, losing, or drawing for the player that moves. Nowadays it is known the outcome of a chess position with at most 7 pieces on board since 2012. (Nalimov Tablebases - Chessprogramming Wiki.)



Win in 31

Move	**Value**
Kd5-e4	Win in 31
Kd5-d4	Win in 36
Kd5-d6	Draw
Kd5-c4	Draw
Kd5-c6	Draw
Kd5-e6	Draw
Qc1-g5	Lose in 7

Figure 17. Endgame database with 6 pieces on board. (Chess Endgame Database - Shredder Chess.)

3.3 Neural Network approach

When AlphaZero won against Stockfish it revolutionized the world of computational chess, not only because it was proclaimed the best chess player in history, but also it did it in a new way, very different from how computer engines were previously programmed. AlphaZero uses neural networks to make extremely advanced evaluations of positions which negates the need to look for 70 million positions per second like Stockfish, it calculated around 80.000 positions per second. DeepMind, the company that develops AlphaZero, stated that AlphaZero reached the benchmarks to defeat Stockfish in a mere four hours. Instead of the usual alpha-beta search algorithm with domain-specific enhancements that other engines use. AlphaZero uses a general-purpose Monte Carlo tree search. (Silver et al., 2018.)

3.3.1 Monte Carlo tree search (MCTS)

The Monte Carlo tree search is another way to tackle the problem with the search function in chess. Monte Carlo in a computing context means that something arbitrary happens. In chess, a module that uses pure MCTS will evaluate the position generating a different sequence of moves from the given position in an arbitrary manner and averaging the final results (win/draw/loss) that it generates. To select which node to keep exploring AlphaZero starts from the root node and uses the following function to calculate the upper confidence bound of the next node. (Simple Alpha Zero, 2017.)

$$U(s,a) = Q(s,a) + c_{puct} \cdot P(s,a) \cdot \frac{\sqrt{\Sigma_b N(s,b)}}{1 + N(s,a)}$$

Figure 18. The function that AlphaZero uses to calculate the upper confidence bound.

This function will prioritize the nodes that form the moment, are considered the best for having led to better results. This selection process will continue recursively for all the nodes in the tree until a node that has not been expanded upon is reached. After this process, the node child nodes are calculated the engine will pick a random child node since the upper bound function will return a random result for every child node. After randomly selecting the child there will be a random simulation of the game where that move was played. This is why the algorithm gets the Monte Carlo name. Even though the game is not truly random, there have been different heuristics that will take more time for the computer to process but will lead to better results. Finally, the algorithm enters the backpropagation where it updates the value of the previous nodes by having taken into account the result of the last simulation. (Cinnéide,)

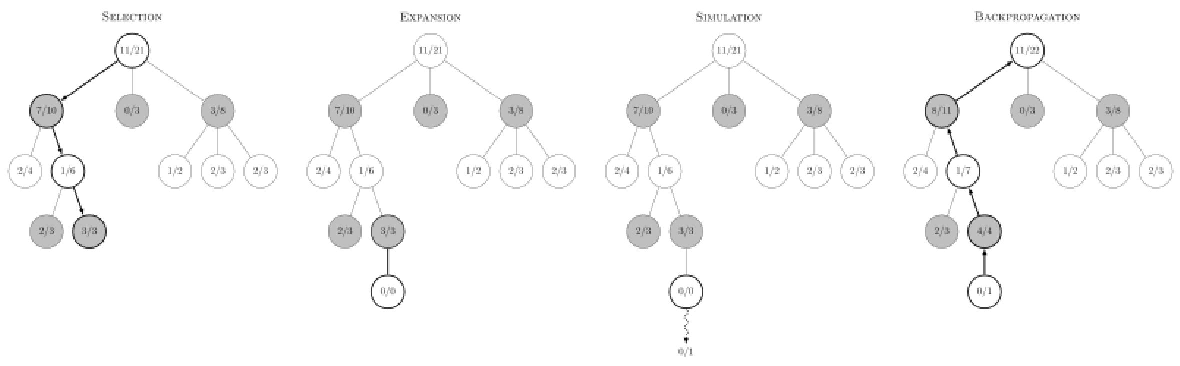


Figure 19. Example of 1 iteration of the Monte Carlo Search Tree (Monte Carlo Tree Search, 2021)

3.3.2 Benefits of using Neural Networks

Using neural networks will bring different advantages over conventional chess engines is the use of Graphical Processing Units (GPUs). With the use of neural networks, AlphaZero can take advantage of the parallel computing power that the GPUs provide. On the other hand, engines like Stockfish were stuck with only using CPU power which is slower since the operations that the engine used were good on a sequence that is where CPUs excel. AlphaZero uses specialized GPUs called TPUs that are google designed hardware to optimize the operation of neural networks. That means adding an additional dimension to the work being done. While GPUs are designed to essentially do huge amounts of parallel arithmetic and trigonometry, TPUs are optimized to rapidly do huge amounts of Matrix multiplication, the fundamental mathematical abstraction of what neurons do. Nowadays, with the latest advancements of Stockfish, they are using a hybrid between conventional chess engines and neural networks so it can benefit from both. (Chess.com)

One of the other benefits of neural networks is the adaptability that it has. The engine only knows the rules of chess and does not have hardcoded rules that humans have made up after years of analysis and experience, so, it becomes easy when you want the engine to learn different variants of chess. This can be useful to find new ways to make chess engaging and learn the best way of playing. There was a paper published by DeepMind about different chess variants, and how the different rules that they added to chess changed the value of certain pieces or how favored was white to win over black. This would be a harder task to accomplish with conventional chess engines since change a few rules of chess can dramatically change the optimal way of playing and all the years of analysis will become obsolete. (Tomašev et ai, 2020.)

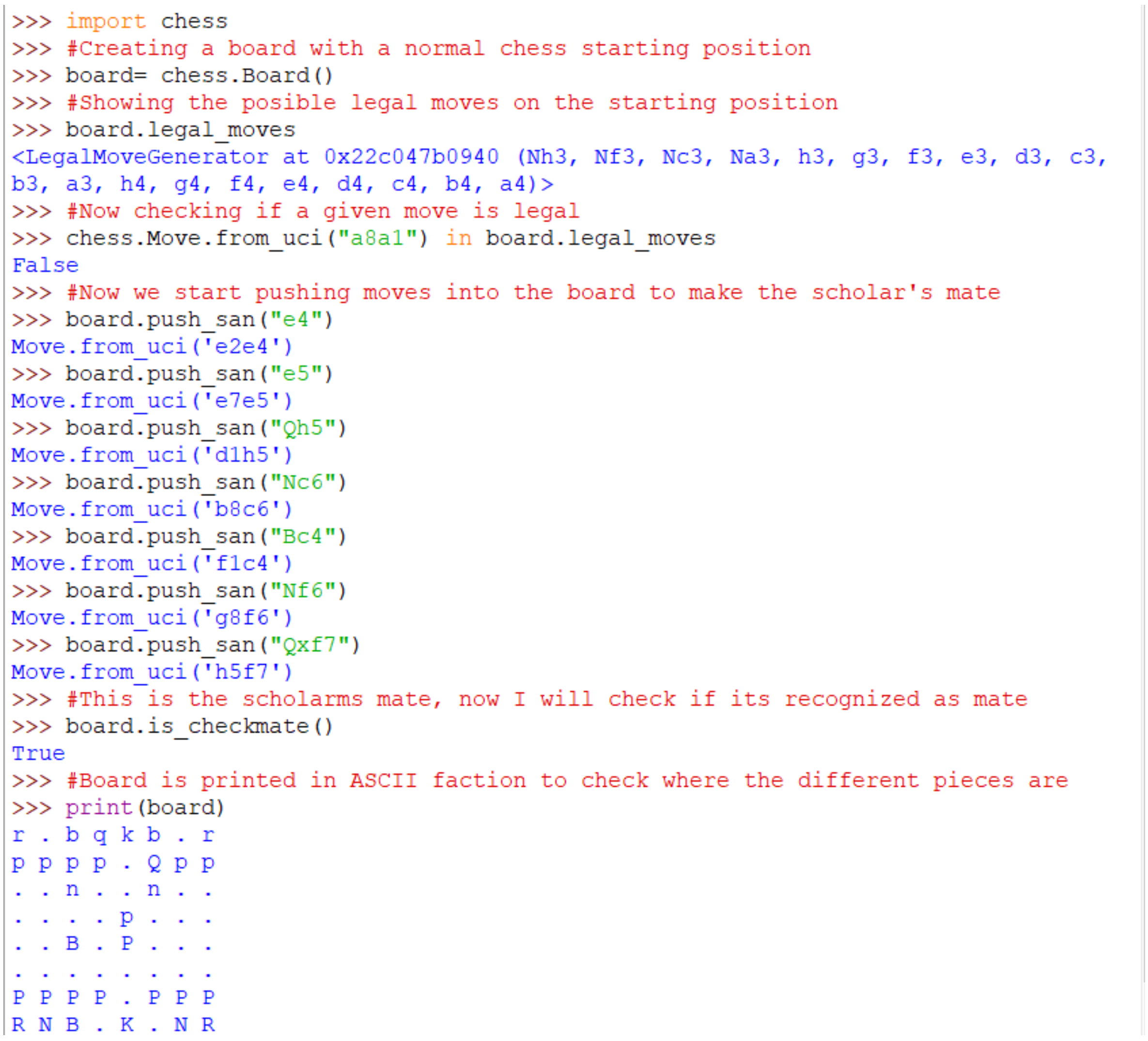
4 Case: Chess engine development

4.1 Introduction

A chess engine is going to be developed with the goal of it having a low-medium level of chess. Like we have seen before there are several approaches on how to tackle this problem. The purpose of this project is not to build the best chess engine, it is to build a low-medium level engine that people can enjoy and observe the changes of its behavior when new functionalities to the search and evaluation functions are added. For achieving this goal, we must take into account the speed of the engine and the computing power needed for it to work properly. Therefore, I have chosen to develop this chess engine using a traditional approach since using the neural network approach will need more computing power and will take longer to process the different moves.

4.2 Programming

For developing this chess engine, I used Python. This programming language is not the fastest one, but it has a lot of documentation and modules that made the development of this project much easier and smooth. Within python the python-chess library was used, this library already has the movement generator, board, and interface already implemented. For starters, I developed the scholar’s mate using the python chess library to learn the basics.



```
>>> import chess
>>> #Creating a board with a normal chess starting position
>>> board= chess.Board()
>>> #Showing the posible legal moves on the starting position
>>> board.legal_moves
<LegalMoveGenerator at 0x22c047b0940 (Nh3, Nf3, Nc3, Na3, h3, g3, f3, e3, d3, c3,
b3, a3, h4, g4, f4, e4, d4, c4, b4, a4)>
>>> #Now checking if a given move is legal
>>> chess.Move.from_uci("a8a1") in board.legal_moves
False
>>> #Now we start pushing moves into the board to make the scholar's mate
>>> board.push_san("e4")
Move.from_uci('e2e4')
>>> board.push_san("e5")
Move.from_uci('e7e5')
>>> board.push_san("Qh5")
Move.from_uci('d1h5')
>>> board.push_san("Nc6")
Move.from_uci('b8c6')
>>> board.push_san("Bc4")
Move.from_uci('f1c4')
>>> board.push_san("Nf6")
Move.from_uci('g8f6')
>>> board.push_san("Qxf7")
Move.from_uci('h5f7')
>>> #This is the scholarms mate, now I will check if its recognized as mate
>>> board.is_checkmate()
True
>>> #Board is printed in ASCII faction to check where the different pieces are
>>> print(board)
r . b q k b . r
p p p p . Q p p
. . n . . n . .
. . . . p . . .
. . B . P . . .
. . . . . . . .
P P P P . P P P
R N B . K . N R
```

Figure 20. Implementation of scholar’s mate using python-chess.

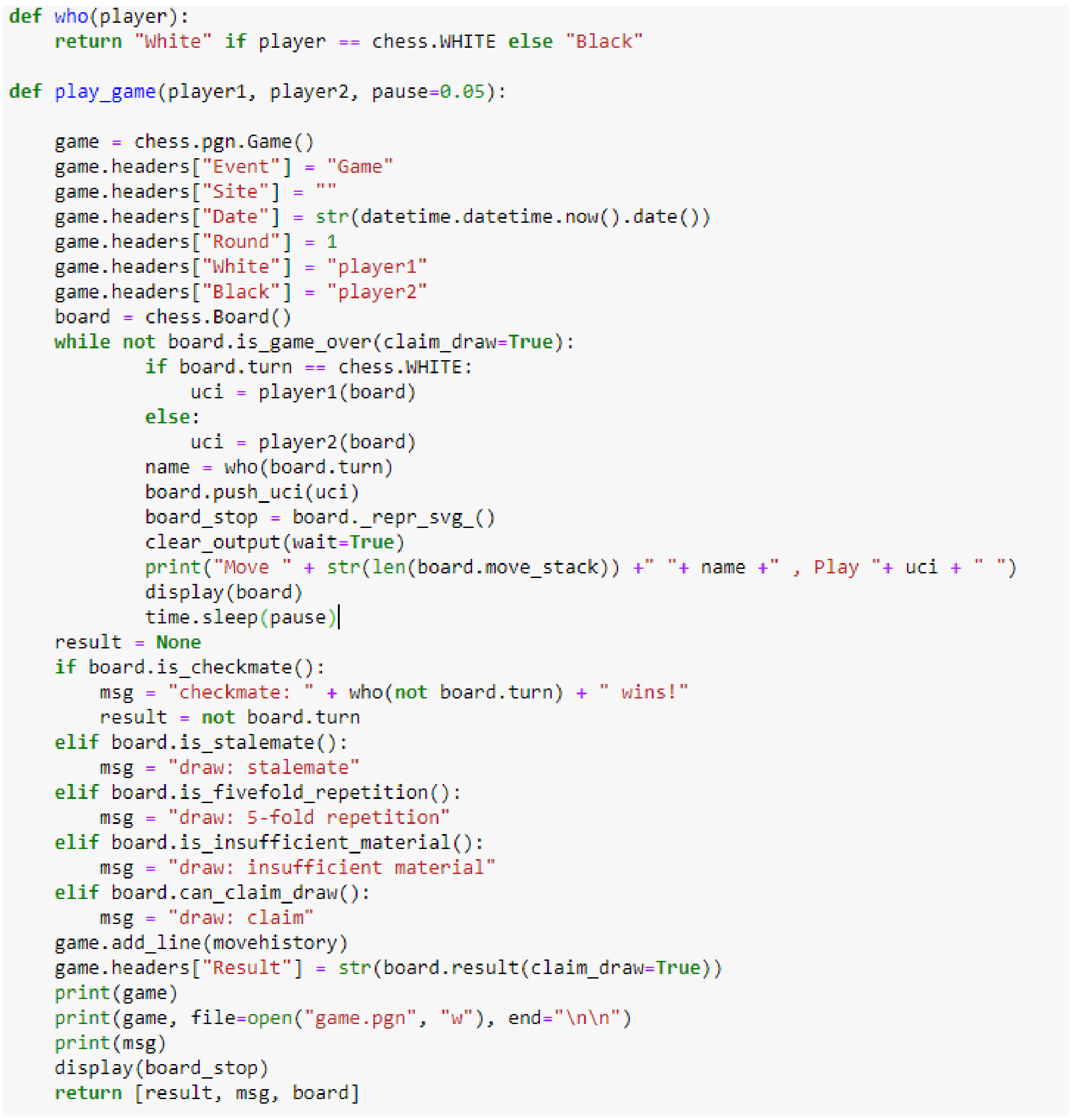
This is a basic implementation of scholar’s mate. For the board to be displayed on anything else than ASCII I made use of Jupyter Notebook, this will help with the visualization of the board. With the python chess library, it is possible to make an SVG board that Jupyter notebook can display successfully to enhance the visualization of this project.

For starters, I programmed a player that made random moves on the chessboard, to test if the first iteration of the chess engine would beat it:

```
def ran_player(board):
    move = random.choice(list(board.legal_moves))
    return move.uci()
```

Figure 21. Random player implementation.

After testing this random player was successfully working a function that made possible games between computers had to be implemented and the code in figure 21 was implemented. This code makes will show the final board and the number of moves that the chess game took. It will display a message saying the result of the game and the number of moves it took, also it will display in SVG the final board and a PGN file with the moves played in the given game.

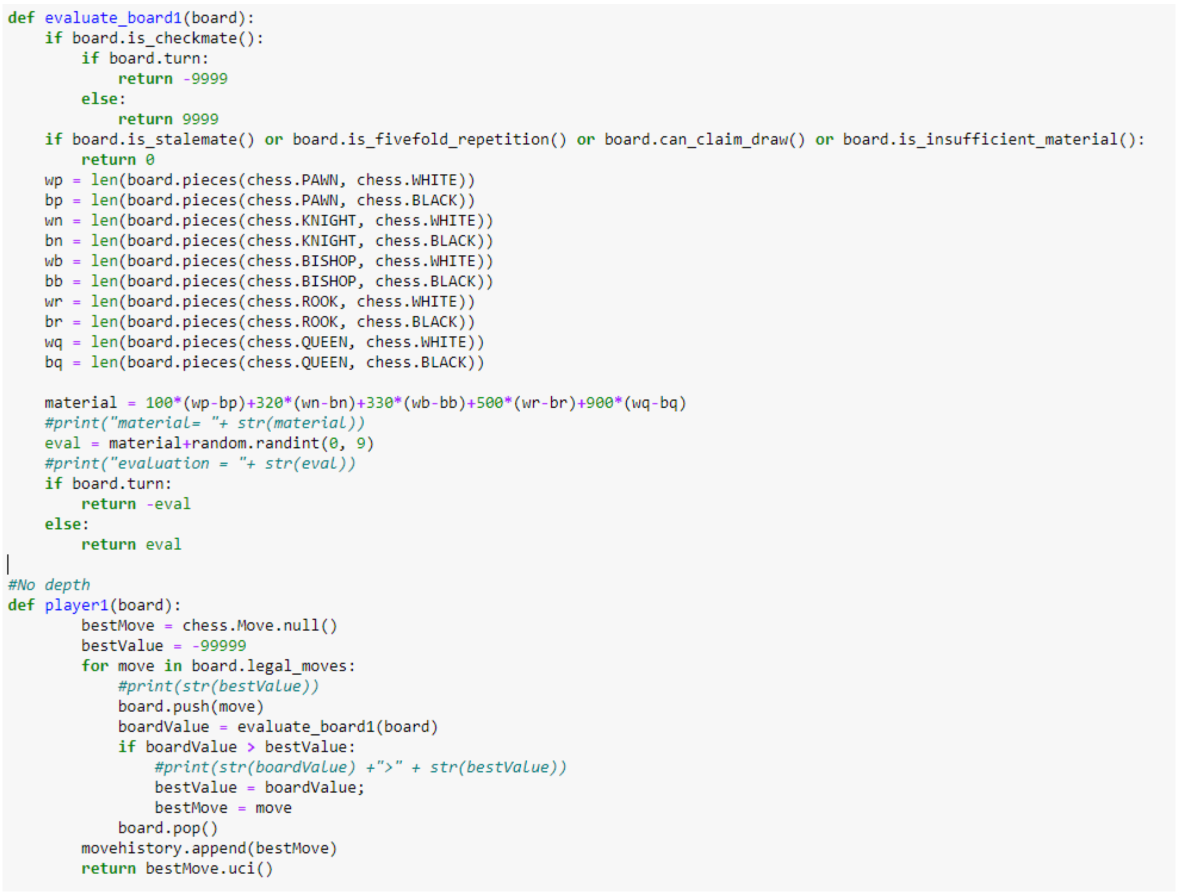


```
def who(player):
    return "White" if player == chess.WHITE else "Black"

def play_game(player1, player2, pause=0.05):

    game = chess.pgn.Game()
    game.headers["Event"] = "Game"
    game.headers["Site"] = ""
    game.headers["Date"] = str(datetime.datetime.now().date())
    game.headers["Round"] = 1
    game.headers["White"] = "player1"
    game.headers["Black"] = "player2"
    board = chess.Board()
    while not board.is_game_over(claim_draw=True):
            if board.turn == chess.WHITE:
                uci = player1(board)
            else:
                uci = player2(board)
            name = who(board.turn)
            board.push_uci(uci)
            board_stop = board._repr_svg_()
            clear_output(wait=True)
            print("Move " + str(len(board.move_stack)) +" "+ name +" , Play "+ uci + " ")
            display(board)
            time.sleep(pause)
    result = None
    if board.is_checkmate():
        msg = "checkmate: " + who(not board.turn) + " wins!"
        result = not board.turn
    elif board.is_stalemate():
        msg = "draw: stalemate"
    elif board.is_fivefold_repetition():
        msg = "draw: 5-fold repetition"
    elif board.is_insufficient_material():
        msg = "draw: insufficient material"
    elif board.can_claim_draw():
        msg = "draw: claim"
    game.add_line(movehistory)
    game.headers["Result"] = str(board.result(claim_draw=True))
    print(game)
    print(game, file=open("game.pgn", "w"), end="\n\n")
    print(msg)
    display(board_stop)
    return [result, msg, board]
```

Figure 22. Play and Who function.

Furthermore, the option to have a human play was also implemented, with the caveat that the human will have to enter the moves in UCI fashion and can't drag the pieces from the chessboard. This can be something that can be explored in the future development of the project.

After all of this, the first iteration of the chess engine was created. It was a simple engine that only counted material on the board. It did not take into account any other things, just what pieces were on the board and it has a value assigned to the different pieces. The values assigned in these engines are the ones Tomasz Michniewski proposes for a simple evaluation function, these values are: Pawn = 100 Knight = 320 Bishop = 330 Rook= 500 Queen = 900, and the function will return -9999 or 9999 there is checkmate on the board.

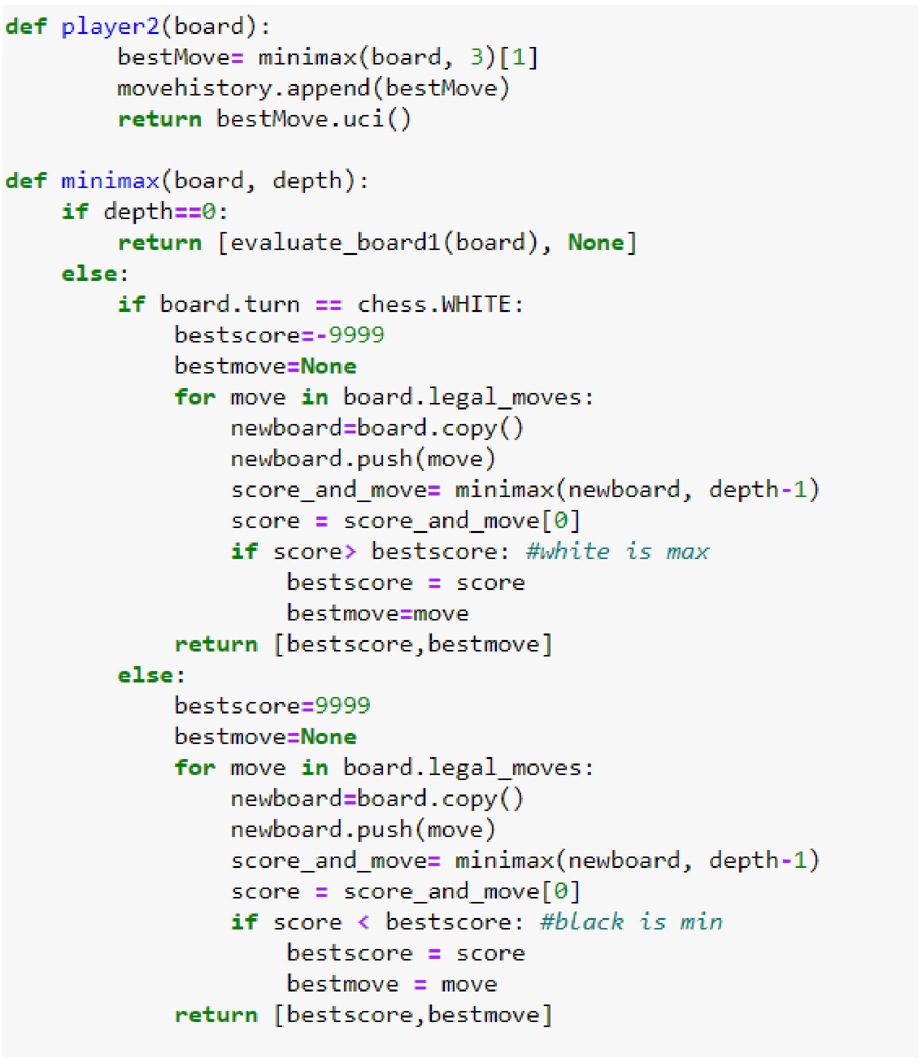


```
def evaluate_board1(board):
    if board.is_checkmate():
        if board.turn:
            return -9999
        else:
            return 9999
    if board.is_stalemate() or board.is_fivefold_repetition() or board.can_claim_draw() or board.is_insufficient_material():
        return 0
    wp = len(board.pieces(chess.PAWN, chess.WHITE))
    bp = len(board.pieces(chess.PAWN, chess.BLACK))
    wn = len(board.pieces(chess.KNIGHT, chess.WHITE))
    bn = len(board.pieces(chess.KNIGHT, chess.BLACK))
    wb = len(board.pieces(chess.BISHOP, chess.WHITE))
    bb = len(board.pieces(chess.BISHOP, chess.BLACK))
    wr = len(board.pieces(chess.ROOK, chess.WHITE))
    br = len(board.pieces(chess.ROOK, chess.BLACK))
    wq = len(board.pieces(chess.QUEEN, chess.WHITE))
    bq = len(board.pieces(chess.QUEEN, chess.BLACK))

    material = 100*(wp-bp)+320*(wn-bn)+330*(wb-bb)+500*(wr-br)+900*(wq-bq)
    #print("material= "+ str(material))
    eval = material+random.randint(0, 9)
    #print("evaluation = "+ str(eval))
    if board.turn:
        return -eval
    else:
        return eval

#No depth
def player1(board):
        bestMove = chess.Move.null()
        bestValue = -99999
        for move in board.legal_moves:
            #print(str(bestValue))
            board.push(move)
            boardValue = evaluate_board1(board)
            if boardValue > bestValue:
                #print(str(boardValue) +">" + str(bestValue))
                bestValue = boardValue;
                bestMove = move
            board.pop()
        movehistory.append(bestMove)
        return bestMove.uci()
```

Figure 23. Code for the first evaluation function and search function.

Now, the engine can look one move ahead and if it sees the possibility of capturing a piece it will do it. This iteration of the engine is still bad since it will not look further than just 1 move and will capture pieces aimlessly, without considering if the piece we just capture leaves creates any weaknesses in the position. Nevertheless, this engine still plays better than the random player implemented earlier. The games often end in draws since the engine has no knowledge of endgames and when the enemy king is the only piece left, it just makes random moves since it does not have a piece to capture, often leading to draw by repetition or stalemates.

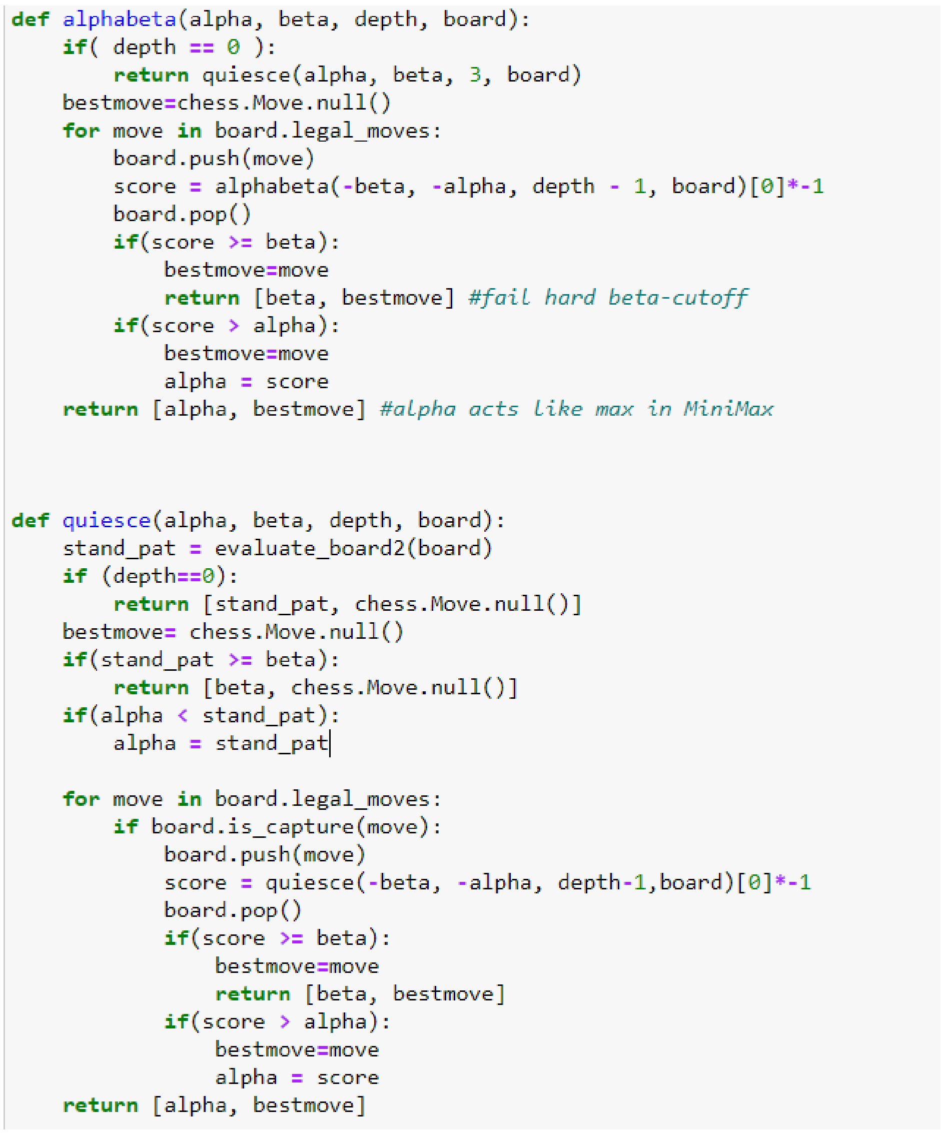
For these reasons, the next step of this engine is to integrate a minimax search so it can look deeper, spot potential traps, and see when you have mate in a given depth. This implementation came with great improvements in play but, doing the minimax with a depth of 3 was consuming too much time per move, I decided to add a timer that tells me how much time it takes for the minimax algorithm to come up with a move and, with a depth of 3, in the starting positions only took around 5-6 second to find a move. But, in the midgame where there are more pieces developed and more possibilities of moves it can take more than 1 minute per move.



```
def player2(board):
        bestMove= minimax(board, 3)[1]
        movehistory.append(bestMove)
        return bestMove.uci()

def minimax(board, depth):
    if depth==0:
        return [evaluate_board1(board), None]
    else:
        if board.turn == chess.WHITE:
            bestscore=-9999
            bestmove=None
            for move in board.legal_moves:
                newboard=board.copy()
                newboard.push(move)
                score_and_move= minimax(newboard, depth-1)
                score = score_and_move[0]
                if score> bestscore: #white is max
                    bestscore = score
                    bestmove=move
            return [bestscore,bestmove]
        else:
            bestscore=9999
            bestmove=None
            for move in board.legal_moves:
                newboard=board.copy()
                newboard.push(move)
                score_and_move= minimax(newboard, depth-1)
                score = score_and_move[0]
                if score < bestscore: #black is min
                    bestscore = score
                    bestmove = move
            return [bestscore,bestmove]
```

Figure 24. Minimax implementation.

As it was discussed earlier in “Search Function” the minimax algorithm takes really long to calculate this amount of data. Therefore, the next step I took to improve this engine is to add the alpha-beta pruning. In theory, this should massively reduce the time and computing cost for finding the best move. To implement the Alpha-beta pruning I chose the Negamax implementation. Also, the quiescence search is integrated to avoid cases where the horizon effect might happen. So now, the engine is using alpha-beta pruning and when it is on the max depth it further looks for possible captures to avoid the horizon effect.

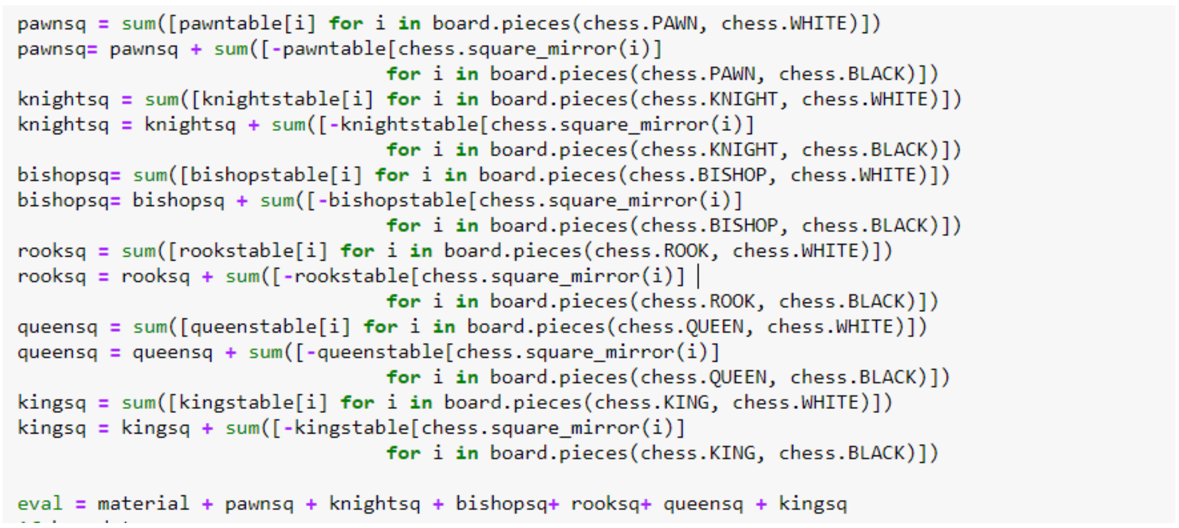


```python
def alphabeta(alpha, beta, depth, board):
    if( depth == 0 ):
        return quiesce(alpha, beta, 3, board)
    bestmove=chess.Move.null()
    for move in board.legal_moves:
        board.push(move)
        score = alphabeta(-beta, -alpha, depth - 1, board)[0]*-1
        board.pop()
        if(score >= beta):
            bestmove=move
            return [beta, bestmove] #fail hard beta-cutoff
        if(score > alpha):
            bestmove=move
            alpha = score
    return [alpha, bestmove] #alpha acts like max in MiniMax



def quiesce(alpha, beta, depth, board):
    stand_pat = evaluate_board2(board)
    if (depth==0):
        return [stand_pat, chess.Move.null()]
    bestmove= chess.Move.null()
    if(stand_pat >= beta):
        return [beta, chess.Move.null()]
    if(alpha < stand_pat):
        alpha = stand_pat

    for move in board.legal_moves:
        if board.is_capture(move):
            board.push(move)
            score = quiesce(-beta, -alpha, depth-1,board)[0]*-1
            board.pop()
            if(score >= beta):
                bestmove=move
                return [beta, bestmove]
            if(score > alpha):
                bestmove=move
                alpha = score
    return [alpha, bestmove]
```

Figure 25. Implementation of Negamax and quiescence search.

Even with all of these new improvements, the engine is still weak. In the first stages of the game, it does not make good decisions since it can´t look very deep into the future, for this reason. I decided it was time to add piece square tables to the evaluation function so the engine will start seeing the benefits of developing pieces when there are no possible captures. This should help with the first stages of the game and help the engine get to the middle game, where it is the strongest. The piece -square tables that I added encourage pieces to perform different tasks.

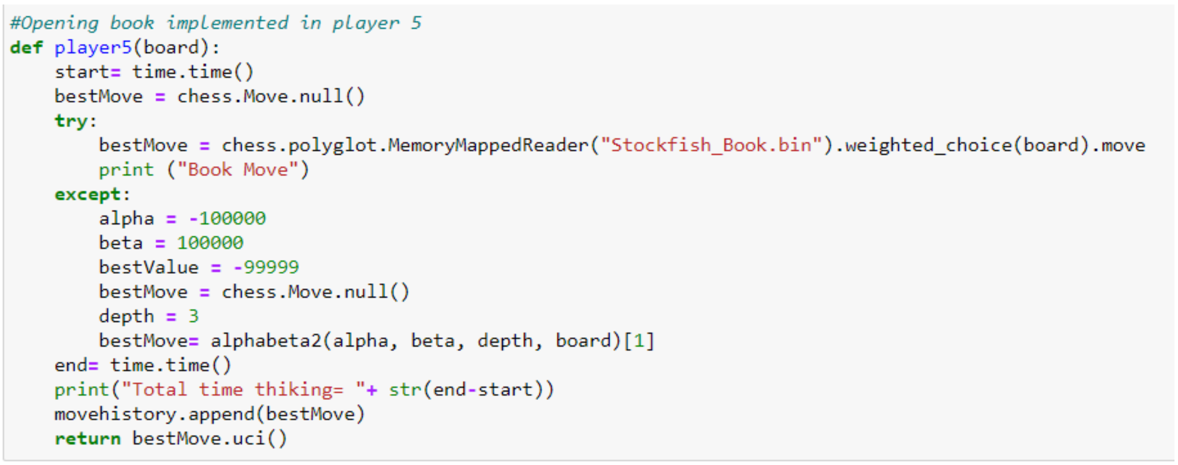
For pawns, the engine is rewarded by pushing them forward and it is discouraged to leave the central pawns unmoved. Knights get better scores when they move towards the center, bishops are incentivized to stay in long diagonals while rooks prefer to infiltrate to the seventh rank while avoiding the a and h files. Queen also prefers to move to the center than stay on corners, since it will help it cover more squares. Right now these square-piece tables are static, they do not vary with the state of the game, so they are pretty general.



```
pawnsq = sum([pawntable[i] for i in board.pieces(chess.PAWN, chess.WHITE)])
pawnsq= pawnsq + sum([-pawntable[chess.square_mirror(i)]
                                for i in board.pieces(chess.PAWN, chess.BLACK)])
knightsq = sum([knightstable[i] for i in board.pieces(chess.KNIGHT, chess.WHITE)])
knightsq = knightsq + sum([-knightstable[chess.square_mirror(i)]
                                for i in board.pieces(chess.KNIGHT, chess.BLACK)])
bishopsq= sum([bishopstable[i] for i in board.pieces(chess.BISHOP, chess.WHITE)])
bishopsq= bishopsq + sum([-bishopstable[chess.square_mirror(i)]
                                for i in board.pieces(chess.BISHOP, chess.BLACK)])
rooksq = sum([rookstable[i] for i in board.pieces(chess.ROOK, chess.WHITE)])
rooksq = rooksq + sum([-rookstable[chess.square_mirror(i)]
                                for i in board.pieces(chess.ROOK, chess.BLACK)])
queensq = sum([queenstable[i] for i in board.pieces(chess.QUEEN, chess.WHITE)])
queensq = queensq + sum([-queenstable[chess.square_mirror(i)]
                                for i in board.pieces(chess.QUEEN, chess.BLACK)])
kingsq = sum([kingstable[i] for i in board.pieces(chess.KING, chess.WHITE)])
kingsq = kingsq + sum([-kingstable[chess.square_mirror(i)]
                                for i in board.pieces(chess.KING, chess.BLACK)])

eval = material + pawnsq + knightsq + bishopsq+ rooksq+ queensq + kingsq
```

Figure 26. Implementation of the piece-square tables in the evaluation function.

The engine still struggles in the early game and tends to make the same moves every match since it cannot be executed at high depths and always follows the same piece-square tables. Therefore, the next step to improve this engine is to add an opening book that it can use to help with the early stages of the game. This opening book will be implemented using the Polyglot format because the python-chess has a function that can make use of this format without trouble. The opening book of choice will be one approved by Stockfish. For now, the engine will look for positions inside the book and if the position is found, the engine will make a random weighted choice between all the different moves. If the position reached is not in the book, the engine will continue to perform as it did before with the old evaluation function and alpha-beta search tree.



```python
#Opening book implemented in player 5
def player5(board):
    start= time.time()
    bestMove = chess.Move.null()
    try:
        bestMove = chess.polyglot.MemoryMappedReader("Stockfish_Book.bin").weighted_choice(board).move
        print ("Book Move")
    except:
        alpha = -100000
        beta = 100000
        bestValue = -99999
        bestMove = chess.Move.null()
        depth = 3
        bestMove= alphabeta2(alpha, beta, depth, board)[1]
    end= time.time()
    print("Total time thiking= "+ str(end-start))
    movehistory.append(bestMove)
    return bestMove.uci()
```

Figure 27. Implementation of the Opening Book in the chess engine.

4.3 Analysis of games played.

Now I will show and analyze Some test games between different engine strengths. With these different tests, we can see if the engine is responding as expected to the new improvements. Every game will include the player that played with each color, the result of the game, the moves that were played in PGN format so everyone can check how the game developed, for example, in websites like lichess.org, and a visual representation of the board when the game ended.

4.3.1 Game 1

[White "player1"]

[Black "ran_player"]

[Result "1/2-1/2"]

1. Nh3 c6 2. Ng5 Qb6 3. Nxh7 Na6 4. Nxf8 Rb8 5. Nxd7 Rh3 6. Nxb6 e6 7. gxh3 g6 8. Nxc8 g5 9. Nxa7 g4 10. Nxc6 Nh6 11. Nxb8 Kd8 12. Nxa6 f6 13. hxg4 Nxg4 14. Nb8 e5 15. Nd7 Nh6 16. Nxf6 Kc7 17. Ng8 Nxg8 18. Rg1 Ne7 19. Rg8 Nd5 20. Rh8 Nc3 21. Nxc3 Kb6 22. Rg8 Ka5 23. Rh8 Kb6 24. Rg8 Ka7 25. Rf8 Ka6 26. Rg8 Ka5 27. Rf8 b6 28. Rh8 Ka6 29.

Rg8 Ka7 30. Rh8 e4 31. Nxe4 Kb7 32. Rg8 Kc7 33. Rh8 b5 34. Rg8 Kb7 35. Rh8 Kc6 36. Rg8 Kb7 37. Rh8 Ka7 38. Rf8 Kb6 39. Rh8 b4 40. Rg8 Kc7 41. Rh8 Kd7 42. Rg8 Ke6 43. Rh8 b3 44. cxb3 Ke7 45. Rg8 Kf7 46. Rh8 Kg6 47. Rg8+ Kh5 48. Rh8+ Kg4 49. Rg8+ Kf4 50. Rh8 Kf5 51. Rg8 Ke6 52. Rh8 Kd5 53. Rg8 Kc6 54. Rh8 Kd5 55. Rg8 Ke6 1/2-1/2

draw: claim

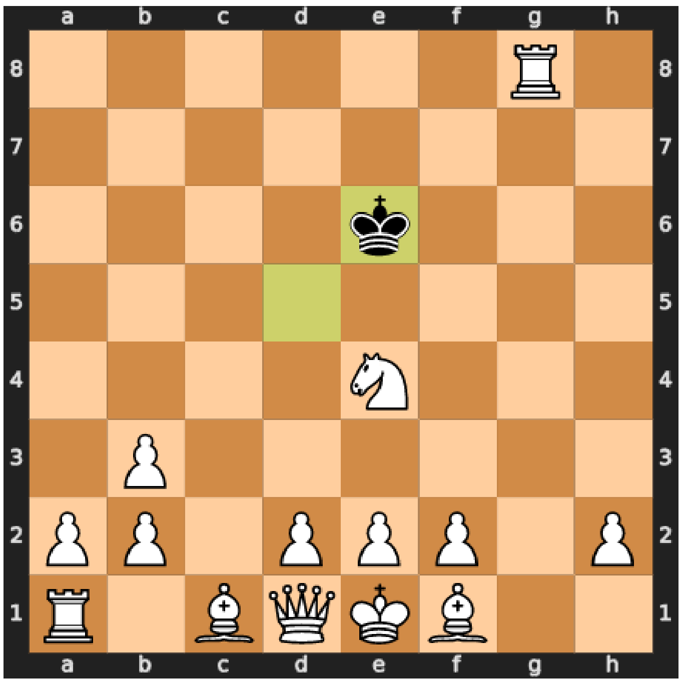


Figure 28. Test game 1.

From this game, we can observe how the first iteration of the engine is working as intended. Every time “player1” could capture a piece it will do it. While the random player just kept moving pieces aimlessly, it did not punish when white took a piece but left one of their own unprotected. In the endgame, “player 1” could not find a mating pattern since it only looks for possible captures, so the game ended in a draw with a 3-fold repetition.

4.3.2 Game 2

[White "player2"] (depth 3)

[Black "ran_player"]

[Result "1-0"]

1. Nh3 Nh6 2. Ng5 Nc6 3. Rg1 d5 4. Rh1 a6 5. Rg1 Nb4 6. Rh1 g6 7. Rg1 Ra7 8. Rh1 Be6 9. c3 b6 10. cxb4 f5 11. Nxe6 Qa8 12. Rg1 Qc8 13. Qa4+ c6 14. Nxf8 Qd7 15. Nxd7 Rxd7 16. Qxc6 Ng4 17. Qc8+ Kf7 18. Qxh8 Nxf2 19. Qxh7+ Ke8 20. Qg8# 1-0

checkmate: White wins!

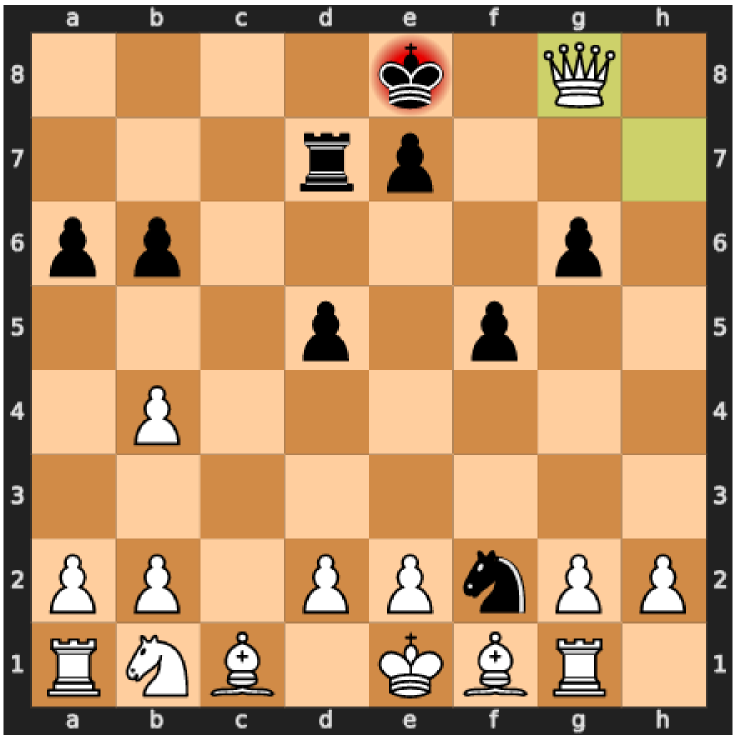


Figure 29. Test game 2.

This game displays interesting behavior. We can see that the engine if it does not see an improvement in the next 4 moves, it will just pick the last move from the movement generator list that does not make the position worse, from moves 4-8 since black did not have any pieces developed and the algorithm cannot see really deep it only moved the rook aimlessly from h1 to g1, until black left the knight in b4 undefended. The engine saw it and attacked the knight on the next move. Also, with the white knight on g5, it could capture the bishop on e6 for a long time, but the white waited till black blundered and left the bishop unprotected to capture it. The engine was also able to win the game unlike the previous iteration, the mate was found by having a very good position and not by creating mating patterns.

4.3.3 Game 3

[White "player2"] (depth 3)

[Black "player1 "]

[Result "1/2-1/2"]

1.Nh3 Nh6 2. Ng5 Rg8 3. Nxh7 Rh8 4. Nxf8 Rxf8 5. Rg1 Rh8 6. Rh1 Rg8 7. Rg1 Rh8 8. Rh1 Rg8 9. Rg1 1/2-1/2

draw: claim

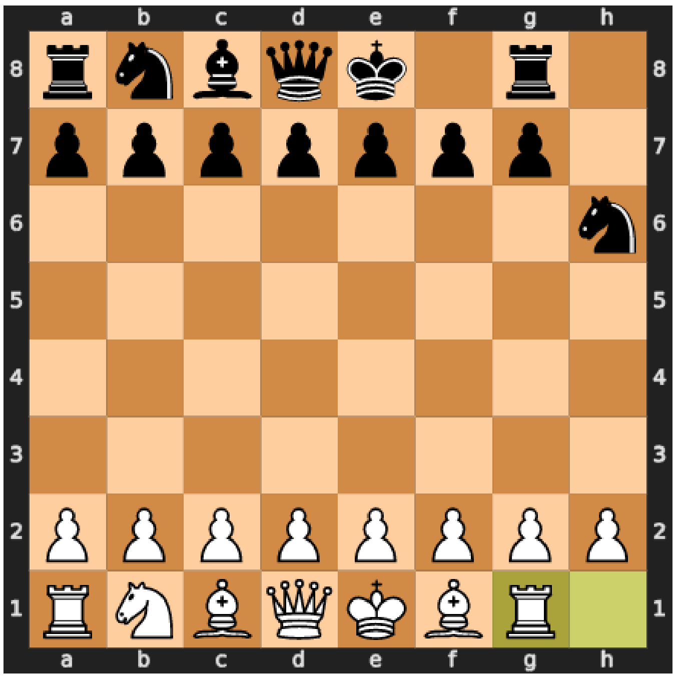


Figure 30. Test game 3.

In this game, we can see one of the main problems with the current engines, when there is no improvement the engine will keep moving the piece over and over causing a 3-fold repetition. This happens because the engine can no see improvements in the position, to solve this problem I changed the evaluation function to add randomly a number between 0 and 9 to the evaluation. This will make it less likely to repeat moves and just keep improving the position we will repeat this match on game 4 with the new improvements to have proper a proper testing environment.

4.3.4 Game 4

[White "player2"] (depth 3)

[Black "player1"]

[Result "1-0"]

1. Nh3 Nc6 2. Ng5 Nf6 3. h4 Ne4 4. Nxe4 Ne5 5. Nc5 f6 6. f4 Nc4 7. e3 Nxb2 8. Bxb2 b5 9. Bxb5 a5 10. Ne6 Ra7 11. a3 Ba6 12. Qh5+ g6 13. Qc5 Bxb5 14. Nxd8 Kxd8 15. Qxa7 Ba6 16. Qxa6 g5 17. Qa8# 1-0

checkmate: White wins!

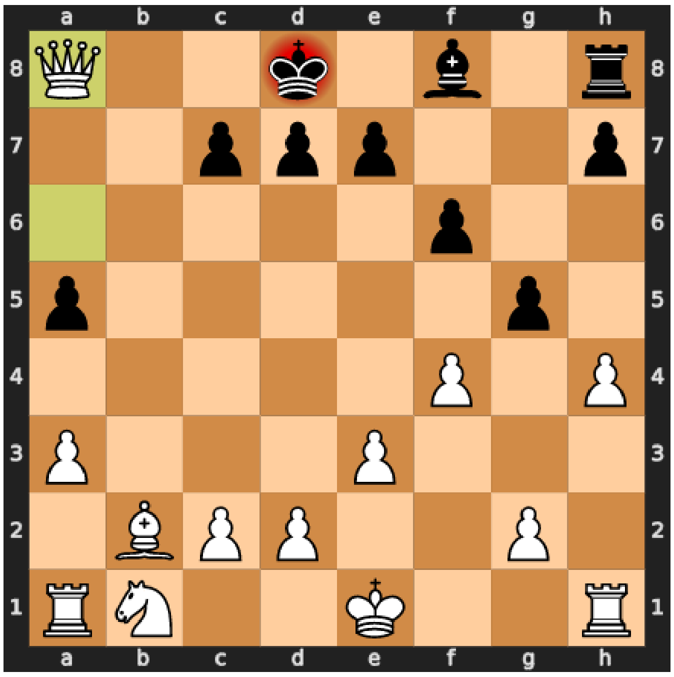


Figure 31. Test game 4.

In this game, we can see how both engines are now working as intended, after tweaking the evaluation function. We can see that white is vastly superior to player black since it can see 3 moves in advance. White never makes a 1-move blunder in this game while black makes several. White capitalizes on these mistakes easily getting into an advantageous position and winning the game with ease.

4.3.5 Game 5

[White "player4"] (depth 3)

[Black "player3"] (depth 3)

[Result "1-0"]

1. Nf3 e6 2. Nc3 Na6 3. e4 h6 4. Bxa6 bxa6 5. O-O Ke7 6. d4 d6 7. e5 dxe5 8. dxe5 Qxd1 9. Rxd1 Bd7 10. Bf4 g5 11. Bg3 Rb8 12. Rab1 Rd8 13. Nd4 Ra8 14. Ne4 Re8 15. Nc5 Bb5 16. Nxb5 Rd8 17. Nxc7 Rxd1+ 18. Rxd1 Bg7 19. N7xa6 Kf8 20. Rd7 h5 21. Rxa7 h4 22. Ra8+ Ke7 23. Ra7+ Kd8 24. Ra8+ Ke7 25. Ra7+ Kf8 26. Nd7+ Ke8 27. Bxh4 Rxh4 28. Nac5 Nh6 29. Nf6+ Kf8 30. Ra8+ Ke7 31. Re8# 1-0

checkmate: White wins!

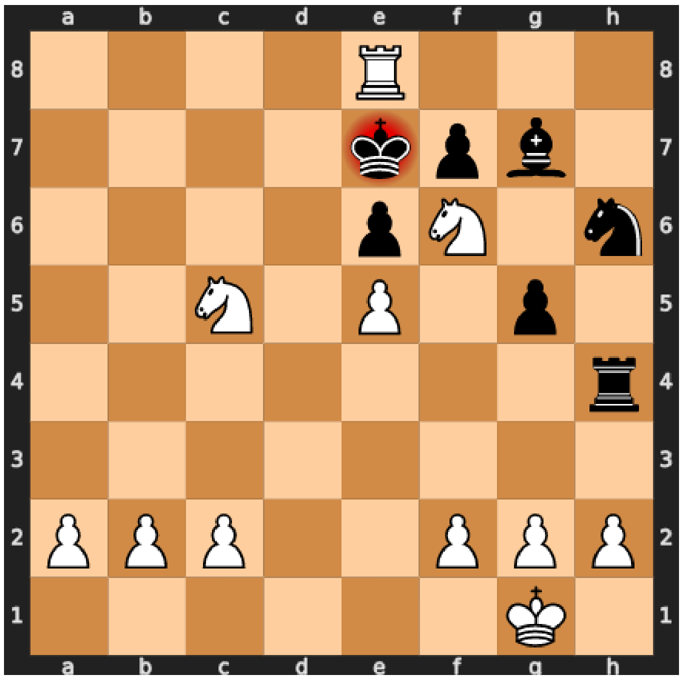


Figure 32. Test game 5

This game is a clear example of why the positioning of the chess pieces is important. The only difference between these engines is the implementation of the piece-square tables. In this game, once white get the king out of the center and all the white pieces start restricting black movement, even if the material is even, white is in a good positional advantage which leads to a clean victory. Another interesting event was when white, in move 27 decided to sacrifice the bishop. This is a clear example that the quiescence search working as intended, the engine recognized the piece was trapped, and instead of mindlessly giving checks to the enemy king to prolong the inevitable of losing the bishop for nothing, it decided to trade the bishop for the pawn.

4.3.6 Game 6

[White "player5"] (depth 3)

[Black "player4"] (depth 3)

[Result "1/2-1/2"]

1. e4 Nf6 2. e5 Ne4 3. d3 Nc5 4. d4 Ne6 5. d5 Nc5 6. Nf3 d6 7. Nc3 dxe5 8. Nxe5 Nbd7 9. Bf4 Nxe5 10. Bxe5 e6 11. Bc4 exd5 12. Nxd5 Ne6 13. Qe2 c6 14. Nc3 Qb6 15. O-O-O Bd7 16. Kb1 O-O-O 17. Ne4 Qa5 18. Bxe6 Bxe6 19. Rxd8+ Qxd8 20. a3 Qd5 21. Bf4 Qa2+ 22. Kc1 Qa1+ 23. Kd2 Qxh1 24. Qf3 Qf1 25. Bg3 c5 26. Be5 Qb5 27. Qc3 Rg8 28. Ke1 Qc6 29. Qf3 Qd5 30. Qf4 c4 31. h3 Bc5 32. Nxc5 Qxc5 33. Kf1 g5 34. Qd4 Qxd4 35. Bxd4 Kb8 36. Be5+ Ka8 37. Kg1 Bf5 38. b3 Re8 39. Bc3 cxb3 40. cxb3 Re2 41. b4 Kb8 42. b5 Bd3 43. Bf6 Re1+ 44. Kh2 h6 45. Bg7 Re6 46. a4 Bc2 47. a5 Bd3 48. Bd4 Bxb5 49. Kg1 Re1+ 50. Kh2 Bc6 51. Bc3 Re7 52. Bd4 Re1 53. Bg7 Re2 54. Bd4 1/2-1/2

draw: claim

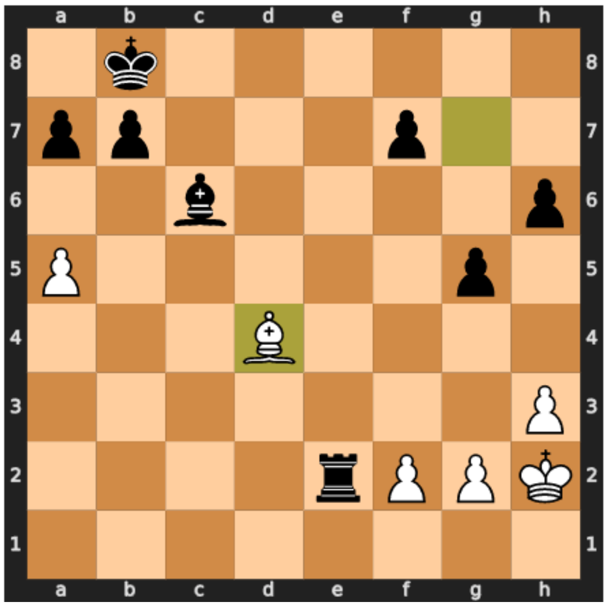


Figure 33. Test game 6.

In this game, after evaluating it with stockfish, we can see that white got a great advantage at the start of the game. This was thanks to the initial book moves which led to a more favorable position. Even when in move 16 white has an advantage of +7.6, according to stockfish 13 with a search depth of 26, the white did not find the winning idea and made some positional blunders. Both engines have the same search and evaluation function after the first opening moves. Some moves in this game took very long to calculate, after further research, I discovered that the quiescence search was searching even more nodes than the normal alpha-beta function. So, I decided to add depth to the quiescence search so the engine does not lose time looking for stupid capturing sequences, from now on the depth of the quiescence search will be 3.

4.3.7 Game 7

[White "player5"] (depth 5)

[Black "player5 "] (depth 4)

[Result "1-0"]

1.Nf3 Nf6 2. g3 d5 3. Bg2 c6 4. c4 g6 5. b3 Bg7 6. Bb2 O-O 7. O-O Qb6 8. Qc2 dxc4 9. Qxc4 Be6 10. Qh4 Nd5 11. Ng5 h6 12. Nxe6 fxe6 13. Bxg7 Kxg7 14. Nc3 Nxc3 15. dxc3 Rf6 16. e3 Na6 17. Qd4 Qxd4 18. cxd4 Rd8 19. Rfd1 Rdf8 20. Rd2 Rd8 21. e4 Rff8 22. Rdd1 Kg8 23. e5 Nc7 24. Rac1 g5 25. Be4 Rfe8 26. Rc5 Nb5 27. Rc4 g4 28. a4 Na3 29. Rcc1 c5 30. Ra1 Nc2 31. Bxc2 cxd4 32. Bg6 Rf8 3. a5 Rd5 34. Be4 Rxe5 35. Rxd4 b5 36. Re1 Rg5 37. Bd3 e5 38. Rde4 b4 39. Bc4+ Kh7 40. Be6 Rd8 41. Rxb4 Rd6 42. Bxg4 Rdg6 43. f3 Rg8 44. Rbe4 Rd8 45. h4 Rg7 46. Rxe5 Kg8 47. h5 Rb8 48. Rxe7 Rxe7 49. Rxe7 Rxb3 50. Be6+ Kf8 51. Rf7+ Kg8 52. Rxa7+ Kf8 53. Bxb3 Ke8 54. Rh7 Kd8 55. a6 Ke8 56. a7 Kd8 57. a8=Q#

checkmate: White wins!

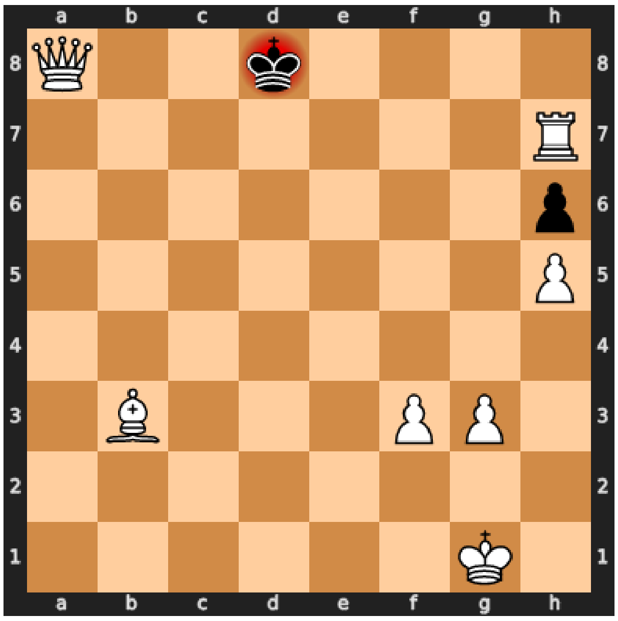


Figure 34. Test game 7

This game is between 2 identical engines but with different depths. The game was equal until move 28 where black had to move the knight from b5. It is very interesting to see that black, with only a depth of 4 makes a blunder since it cannot calculate that moving the knight to a3 will trap it and get captured in the next 5 moves. On the other hand, white, as soon as black trapped the knight it started doing the 5-move sequence that will lead to the knight being capture. After losing the knight black just kept losing pawns and pieces until it got checkmated.

4.3.8 Game 8

[White "player5"] (depth 4)

[Black "human_player"] (Roberto Marrero Rodrígez)

[Result "1-0"]

1.e4 e5 2. Nf3 Nf6 3. d4 exd4 4. Qxd4 Nc6 5. Qd3 Bc5 6. Bg5 O-O 7. Nc3 h6 8. Bh4 g5 9. Bg3 Nb4 10. Qd2 Re8 11. Bb5 Nxe4 12. Nxe4 Rxe4+ 13. Kd1 d5 14. a3 Nc6 15. Bxc6 bxc6 16. Qc3 Bd6 17. Qxc6 g4 18. Bxd6 gxf3 19. Qxa8 fxg2 20. Kd2 gxh1=Q 21. Rxh1 Qxd6 22. Qxc8+ Kh7 23. Qf5+ Qg6 24. Qxd5 Qg2 25. Qxf7+ Kh8 26. Qf8+ Kh7 27. Qf7+ Qg7 28. Qf5+ Qg6 29. Qxg6+ Kxg6 30. Re1 Rxe1 31. Kxe1 Kh5 32. Kf1 Kg4 33. Kg2 h5 34. Kf1 h4 35. Kg2 h3+ 36. Kf1 Kf3 37. Kg1 c5 38. a4 a5 39. Kf1 c4 40. Kg1 Ke2 41. f4 Kd2 42. b3 Kxc2 43. bxc4 Kc3 44. c5 Kc4 45. c6 Kc5 46. c7 Kb4 47. c8=Q Kxa4 48. Qxh3 Kb4 49. f5 a4 50. f6 a3 51. f7 a2 52. f8=Q+ Ka4 53. Qa8+ Kb4 54. Qxa2 Kb5 55. Qhe6 Kb4 56. Qec4#

checkmate: White wins!

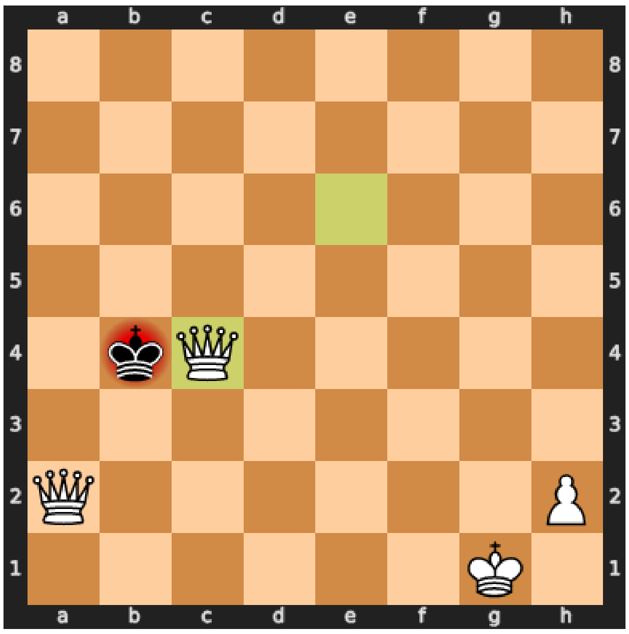


Figure 35. Test game 8

This game I played as black versus the final iteration of the engine in a depth of 4. The human_player function works as intended. The game was interesting. Black managed to get a good advantage in moves 12 to 15. This is due to my quickly castling my king and punishing the opponent for taking his queen out early. Even with an advantage of 7.7 points according to stockfish 13 at move 21, I did not find the winning move. After not finding the correct sequence of moves that kept me in the lead the engine started to make better plays than me and slowly took all my pawns until it eventually checkmated me.

4.4 Conclusion

The engine development process is done for now. The objectives of the case have been reached successfully. The level of play is good. The engine can consistently beat me and I'm on the 60th percentile of players in the chess.com database around 1100-1200 rating. So, we can assess that the engine has a medium-low level of play. On the other hand, between the possibility of visualization and different test games played, it was possible to see how the engine kept evolving and how it improved with each iteration. The engine is still far from the best, one of its main flaws is the time required to come with a move. Right now, there is no move ordering implemented in any way which will help drastically with the speed the Alpha-beta pruning will operate. Other possible solutions for the speed are adding transposition tables and an incremental evaluation function, so the engine does not have to calculate over and over positions that it has already seen.

One of the difficulties encountered while doing the case was learning how to code in a different language like python and how to use Jupyter Notebook. This was my first time using Jupyter and some of the libraries used in this project.

Most of the time in the development process was spent fixing bugs and reassuring that the different algorithms were implemented correctly. There were a lot of bugs when trying to use the different libraries and sometimes, the engine will fail to check if a position is a draw or not, so it will randomly draw the game when it’s ahead on material. To check if the algorithms were implemented correctly, I had to check in multiple manners. To check if the search and evaluation functions were implemented correctly, I had to check the time it took for finding the correct move and compare it to previous iterations of the engine to see if it had improved. And to check if it was not missing good moves or if the evaluation function was implemented successfully, I had to check the engine of similar strength against each other but with different depths. If the engine with a higher depth can consistently beat the one with lower depth the evaluation function and search function have been implemented correctly.

5 Summary

This research was made to create a low-mid level chess engine and analyze how it evolved with the different improvements that were added along the way. With the results achieved, we can conclude that goals were achieved successfully. This does not mean that the engine cannot be further improved and developed. Engine still takes a long to process get the best move on the position. On a depth of 5, it can take up to 30 to 40 minutes to make the move. A move ordering heuristic will be highly recommended to enhance the speed of the engine.

I would like to highlight the game seven. There is a very interesting interaction that I analyze in the chapter 4.3.7 where black gets the knight trapped because it does not have enough depth and white is playing with a higher depth sees the 5-move sequence to capture it. This leave us wondering, if the engine did not take that long to find a move, if we could execute the engine on depths of 10 or 12, like other engines like Stockfish can do without much effort, what will be the true strength of the evaluation function that was implemented.

During the research of this project, I have realized how complex and how much work have been put into developing chess engines and how far away are we from reaching the limit. It always feels like we can make smaller improvements to the engine, and you will never be done. There will be always something to improve in the chess engine world whether it will be through conventional method or neural networks.

References

3Blue1Brown. 2017. Neural networks. Retrieved May 2, 2021, from 3Blue1Brown website: https://www.3blue1brown.com/neural-networks

AKDEMIR, A. *Tuning of Chess Evaluation Function by Using Genetic Algorithms*. Bogaziçi University.

Alpha–beta pruning. 2021. In *Wikipedia*. Retrieved from https://en.wikipedia.org/w/index.php?title=Alpha%E2%80%93beta_pruning&oldid=1020805901

AlphaZero—Chess Engines. Retrieved May 3, 2021, from Chess.com website: https://www.chess.com/terms/alphazero-chess-engine

Beal, D. F. 1982. Benefits of minimax search. In *Advances in computer chess* (pp. 17–24). Elsevier.

Chess Endgame Database—Shredder Chess. Retrieved May 1, 2021, from https://www.shredderchess.com/online/endgame-database.html

Chess Programming Part VI: Evaluation Functions. Retrieved May 5, 2021, from GameDev.net website: http://www.gamedev.net/reference/articles/article1208.asp

Cinnéide (Mel_OCinneide), M. Ó. ¿Cómo juega AlphaZero al ajedrez? Retrieved May 5, 2021, from Chess.com website: https://www.chess.com/es/article/view/como-juega-alphazero-al-ajedrez

CPW-Evaluation. Retrieved April 30, 2021, from https://www.chessprogramming.org/Evaluation

Duan, Y., Edwards, J. S., & Dwivedi, Y. K. Artificial intelligence for decision making in the era of Big Data–evolution, challenges and research agenda. *International Journal of Information Management*, *48*, 63–71.

Evaluation of Pieces—Chessprogramming wiki. Retrieved May 5, 2021, from https://www.chessprogramming.org/Evaluation_of_Pieces

FIDE Ratings. Retrieved April 30, 2021, from https://ratings.fide.com/top_lists.phtml

Fig. 1. Game tree for Tic-Tac-Toe game using MiniMax algorithm. Retrieved April 5, 2021, from ResearchGate website: https://www.researchgate.net/figure/Game-tree-for-Tic-Tac-Toe-game-using-MiniMax-algorithm_fig1_262672371

Hartikka, L. 2017, March 30. A step-by-step guide to building a simple chess AI. Retrieved May 5, 2021, from FreeCodeCamp.org website: https://www.freeco-decamp.org/news/simple-chess-ai-step-by-step-1d55a9266977/

Holcomb, E. Chess Board Positions. Retrieved April 30, 2021, from http://www.nwchess.com/articles/misc/Chess_Board_Positions_article.pdf

Hornyak, T. 2013. Fujitsu supercomputer simulates 1 second of brain activity—CNET. Retrieved May 11, 2021, from https://www.cnet.com/news/fujitsu-supercomputer-sim-ulates-1-second-of-brain-activity/

IBM. 2021, May 10. What is Artificial Intelligence (AI)? Retrieved May 11, 2021, from https://www.ibm.com/cloud/learn/what-is-artificial-intelligence

IBM100—Deep Blue [CTB14]. (2012, March 7). Retrieved April 30, 2021, from http://www-03.ibm.com/ibm/history/ibm100/us/en/icons/deepblue/

Krittanawong, C. 2018. The rise of artificial intelligence and the uncertain future for physicians. *European Journal of Internal Medicine*, *48*, e13–e14.

Kumar, N., & Al, E. 2021. Can Chess Ever Be Solved. *Turkish Journal of Computer and Mathematics Education (TURCOMAT)*, *12*(2), 2814–2819. https://doi.org/10.17762/turcomat.v12i2.2311

Levinson, R., & Weber, R. 2001. Chess Neighborhoods, Function Combination, and Reinforcement Learning. In T. Marsland & I. Frank (Eds.), *Computers and Games* (pp. 133–150). Berlin, Heidelberg: Springer Berlin Heidelberg. https://doi.org/10.1007/3-540-45579-5_9

Material—Chessprogramming wiki. Retrieved May 5, 2021, from https://www.chesspro-gramming.org/Material

Monte Carlo tree search. 2021. In *Wikipedia*. Retrieved from https://en.wikipedia.org/w/in-dex.php?title=Monte_Carlo_tree_search&oldid=1026010523

Nalimov Tablebases—Chessprogramming wiki. Retrieved May 5, 2021, from https://www.chessprogramming.org/Nalimov_Tablebases

Nielsen, M. A. (2015). *Neural Networks and Deep Learning*. Retrieved from http://neural-networksanddeeplearning.com

Opening Book—Chessprogramming wiki. Retrieved May 5, 2021, from https://www.chess-programming.org/Opening_Book

Pawn Structure—Chessprogramming wiki. Retrieved May 5, 2021, from https://www.chessprogramming.org/Pawn_Structure

Pete. (n.d.). AlphaZero Crushes Stockfish In New 1,000-Game Match. Retrieved May 4, 2021, from Chess.com website: https://www.chess.com/news/view/updated-al-phazero-crushes-stockfish-in-new-1-000-game-match

Piece-Square Tables—Chessprogramming wiki. Retrieved May 5, 2021, from https://www.chessprogramming.org/Piece-Square_Tables

Podlesak, M. 2019, January 23. History Of Chess Computer Engines. Retrieved April 28, 2021, from Chessentials website: https://chessentials.com/history-of-chess-com-puter-engines/

Quiescence search. 2021. In *Wikipedia*. Retrieved from https://en.wikipedia.org/w/in-dex.php?title=Quiescence_search&oldid=997971624

Quiescence Search—Chessprogramming wiki. Retrieved May 5, 2021, from https://www.chessprogramming.org/Quiescence_Search

Raschka, S., & Mirjalili, V. 2017. Python Machine Learning: Machine Learning and Deep Learning with Python. *Scikit-Learn, and TensorFlow. Second Edition Ed*.

Russell, S., & Norvig, P. 2002. *Artificial intelligence: A modern approach*.

Schaeffer, J., Powell, P. A. D., & Jonkman, J. 1983. A VLSI Chess Legal Move Generator. In R. Bryant (Ed.), *Third Caltech Conference on Very Large Scale Integration* (pp. 331–350). Berlin, Heidelberg: Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-95432-0_19

Scoones, D. 2007, February 4. Dadian Chess: Horizon effect. Retrieved May 6, 2021, from Dadian Chess website: http://dadianchess.blogspot.com/2007/02/horizon-effect.html

Search—Chessprogramming wiki. Retrieved May 5, 2021, from https://www.chessprogramming.org/Search

Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., … Hassabis, D. (2018). A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science*, *362*(6419), 1140–1144. https://doi.org/10.1126/science.aar6404

Simple Alpha Zero. Retrieved March 2, 2021, from https://web.stanford.edu/~surag/posts/alphazero.html

Team (CHESScom), C. com. Computer Chess Engines: A Quick Guide. Retrieved April 30, 2021, from Chess.com website: https://www.chess.com/article/view/computer-chess-engines

Tempo—Chessprogramming wiki. Retrieved May 5, 2021, from https://www.chessprogramming.org/Tempo

Tesauro, G. 2001. Comparison training of chess evaluation functions. In *Machines that learn to play games* (pp. 117–130).

Tomašev, N., Paquet, U., Hassabis, D., & Kramnik, V. 2020. Assessing Game Balance with AlphaZero: Exploring Alternative Rule Sets in Chess. *ArXiv:2009.04374 [Cs, Stat]*. Retrieved from http://arxiv.org/abs/2009.04374

Turing, A. M. 1950. Computing Machinery and Intelligence. In *Computing Machinery and Intelligence* (p. 22).

Turing Test in AI - Javatpoint. Retrieved May 9, 2021, from Www.javatpoint.com website: https://www.javatpoint.com/turing-test-in-ai