

NEA Project

Jayden Gill

Chess Engine

UTC Reading

NEA – Chess Engine

Contents

[Section 1: Analysis 2](#_Toc161315163)

[Project Description 2](#_Toc161315164)

[Background and Research 2](#_Toc161315165)

[Stockfish 2](#_Toc161315166)

[End User Interview 2](#_Toc161315167)

[What will I Do? 3](#_Toc161315168)

[Section 2: Documented Design 4](#_Toc161315169)

[System Overview 4](#_Toc161315170)

[Chess Architecture 4](#_Toc161315171)

[Candidate Moves 4](#_Toc161315172)

[Basic-Evaluation Function 5](#_Toc161315173)

[Minimax Algorithm 6](#_Toc161315174)

[Hardcoded Chess Concepts 8](#_Toc161315175)

[GUI 9](#_Toc161315176)

[References 9](#_Toc161315177)

# Section 1: Analysis

## Project Description

Chess is not a solved game, by any means. After just three moves, there are over one hundred and twenty million possibilities. So, at this point in time, we don’t possess the processing power, or memory, to compute every single possibility in chess. However, we can look moves into the future, using computers to find the ‘best move’. Now using computers, we can improve ourselves further, by using it to check our games, or playing against it. Therefore, in my project, I will be creating a ‘Chess Engine’ that gives evaluations of positions and output the ‘best move’, as well as being able to play different difficulties of the engine. You will also be able to play against other people on your network, access a leaderboard and many other features.

## Background and Research

The top three chess engines in the world are Stockfish, Komodo Dragon and Fat Fritz (Champion, Top 5 Chess Engines, n.d.). So, I will delve into how stockfish has been coded, and see different techniques that have been used.

### Stockfish

Stockfish uses something called ‘Bitboards’. As a chessboard is made up of 64 squares, the positions of a given piece can be stored in a 64-bit variable. Every bit corresponding to a square. Therefore, if it is set to 1, then a piece is present. This is how stockfish can “see” the board and interact (via binary shifts etc). It will then use these bitboards to find ‘candidate’ moves and store them all in a list.

Now it has a list of ‘candidate’ moves, it can through them and find which leads to the best evaluation by using a minimax algorithm, (which I will delve deeper into later). So, we must have an evaluation function which we want to maximise. So, we must hardcode different chess concepts into the engine, giving ‘rewards’ (points) for good moves and taking away points for bad ones.

So, some basic concepts involve, material, strategy, and space. However, I will go more into depth with this later on.

### End User Interview

Now I will investigate want an end user may want in a chess engine. To collect meaningful feedback from end users, and adjust the project accordingly, I must choose a well-rounded form of user feedback. The main two options are surveys, and interviews. They are both good ways of analysing the detailed requirements and demands of chess enthusiasts. Now a survey is good for getting large amounts of data, and therefore I will get many ideas in which I could expand upon. However, it would not get me the level of detail, that a one-to-one interview would get me. For this reason, I am going to conduct a one-to-one interview.

|  |
| --- |
| Me: I’m working on a chess engine and would love if you could give me so feedback to help make it user friendly and as enjoyable as possible. To start, could you just share a bit about your experience with chess, and then what you want in a chess engine |
| Witek: Sure, I’ve been playing chess for about 5 years now, mainly online due to covid. I use chess engines for a few different things, so I’ve got a few different requirements for each one. I use it to play against, so for this id like one roughly my skill, or a little higher. To give me a challenge, but not so that I don’t have a chance. But I will also use it to review my games, find the best moves in places I was unsure and give me evaluations on positions. So, with this I obviously want the best engine possible, so I can improve my game. Oo, another nice touch would be if it could say the worst moments in my game, the one where the move I made, made the evaluation shift massively, and then providing the alternative moves in the position. |
| Me: Well, you’ve answered quite a few of my questions there, so that helps me massively. Thank you. I’m thinking about adding a leaderboard where people can see who has played the highest difficulty engine and won. What’s your thoughts on this? Also, Are there any specific preferences in regard to the design of the program? |
| Witek: Yes, I think the leaderboard is a cool idea for some people. But to be honest, I probably wouldn't use it, all too often, if ever. As I like to focus on my gameplay, I’d prefer a self-leaderboard, to show the best wins I’ve personally had. In regards the design, just something simple to use. I don’t care too much about it looking all fancy, as long as its self-explanatory, I’m fine. |
| Me: Well, I think you’ve answered all my question Witek. Thank you so much for your help, I’ll try to implement as much as possible. |

Interview Between Myself and Witek

### What will I Do?

I interviewed Witek from my local chess club, “Reading Chess Club” about the project. He has given some very good ideas and points. I am going to explain some things that are going to change in regards the original plan.

The leaderboard is now going to be reflective on the player, and their past wins rather than against others. So that players can focus on their self-improvement rather than comparing to other people. I want to add a “key moments” feature which uses a shift in evaluation to provide key moments in a player’s game. This will make the learning and improvement far easier, as rather than looking through your whole game, trying to fish out the moments you went wrong, the computer can do it for you. Finally, I will include two versions of the engine; One where you can play against a desired level, and another which you can use to analyse your game. So that no matter what you want out of the engine you have the option to do both.

# Section 2: Documented Design

## System Overview

In this section, I will have a quick overview of the system, and include a little more detail into certain features will be added into the program.

### Chess Architecture

The heart of the Chess Engine lies in its sophisticated algorithms, and efficient data structures. But for the chess engine to do anything, it has to “see” the board. Or move technically, be given the information of the whereabouts of all the pieces. We do this using “bitboards”. Chess is played on an 8x8 grid, meaning that we have 64 squares. This can be stored in a 64-bit variable, or a 64 long array. Where a 1 means a piece is in that location and 0 means there is not. This would mean that each piece would also need their own bitboard, and an AND operation would be needed to know where each piece is in the board.

However, bitboards seem like a very long way to do this. Yes, in binary this is the only way! However, in high level programming languages, we have a few alternatives. We can have an array with length 64. Where each piece has a corresponding label, e.g. White King -> WK, Black Knight -> BN. This means that’s we don’t have to have many different bitboards, causing more storage to be used etc.

Upon more research though, there is a far easier way to implement this into my project. Due to the representation of the chess board not being my main goal, I can use existing library to do this for me. Python actually has a library to represent a chess board called.

Python-Chess (python-chess, n.d.). Therefore, I will be using this, as it does not impact performance, and allows me to spend more time on the engine rather than preliminary tasks.

### Candidate Moves

As well as seeing the board, the computer must also understand how the pieces are allowed to move. So that it can analyse the best combinations of moves which lead to the best position. Efficient bitwise operations and logical function are applied to determine legal moves, and these would be added to a list of legal moves.

Now rather than hard coding all of this into my program and taking away time from the technical solution of the chess engine. I can use the features of the library python-chess, which I talked about previously. It has features, in which I can add all the legal moves to an array, which I can use later.

### Basic-Evaluation Function

The basic evaluation function actually isn’t as hard as you may think. If the board is checkmate, and the turn is for white then a score of -1000 is outputted, if its blacks turn then a score of 1000 is outputted. If it is stalemate (no legal moves remaining), then a score of 0 is returned. If none of these conditions are met, then we use some basic chess strategy that is taught when first learning the game. If you have ever studied chess before, you would know that pieces are assigned “values”. Pawns are one point, Knights and Bishops are three points, Rooks are five points, and Queens are nine points. Therefore, we can get the total points for each side, and subtract white’s material points, from blacks. This is the evaluation. If its negative, then black is winning, if it is positive then white is winning.

Figure 1 – Points Scoring System for Evaluation Function

A screenshot of a game

Description automatically generated

A screen shot of a computer program

Description automatically generatedNow, this can be used to make a relatively strong and advanced chess engine. Which would pair well against even some of the stronger amateur players. Which capitalises on the other players mistakes. However, it would need to look very fair into future moves to beat some of the more advanced players.

Figure 2 – Pseudocode for the basic Evaluation Function

### Minimax Algorithm

In the last sub heading we learnt how a computer, comes up with a basic evaluation. But how does a computer pick which move is the best out of the list of available moves, especially if the evaluation is the same for a multitude of moves.

Well, it looks into the future. No, it doesn’t have a time travelling DeLorean, but it can play multitudes of potential moves to find the best evaluation. For example, it will play a move, then it will look at what the other player can do and play the move for them, and then their move and then the other players move etc until the desired “depth” (certain number of moves into the future) is reached. This will then get an evaluation assigned to this “line” (a sequence of moves). It will go from the last move in the line and go back a move and play an alternative move. Once all these possibilities are calculated, it will go back to the previous move in the line, and find all these possible moves etc. So that every “line” (sequence of moves) has been made, and each line has a corresponding evaluation. It will then play the best move, the move with the highest evaluation.

Now we must be careful, we could end on our move believing we have a high evaluation. But on the next move we could get checkmated. As we have not calculated their move. Therefore, we must end all our calculations once we have finish calculating our opponents move, not our own.

So how does the computer do this? Using the minimax algorithm (L, n.d.), a backtracking algorithm used very widely in a multitude of other games, such as Tic-Tac-Toe. It is made up of two “players”, a minimiser and a maximiser. The maximiser tries and gets the highest score possible, whilst the minimizer tries to get the lowest score possible.

A diagram of a structure

Description automatically generated

Figure 3 – Visualization of the Minimax Algorithm

So as shown in the diagram the Maximiser goes first, lets say that the maximiser goes left first. Now it’s the minimisers turn. So the minimiser is going to pick the lowest score, which is two. So the maximum score for the left is two, now we backtrack to the start. The maximiser now goes right. The minimiser will pick the score of one as it is lower than the nine. So now the maximum score for the right is one. So the maximiser will go left as it guarantees a minimum score of 2, over the minimum score of 1.

In real life, maybe the other player will play the move that leads to the nine result, however we want the maximum guaranteed score, not just the maximum score. As this will beat not just amateur players, but advanced players too who make the correct move.

#### Alpha-Beta Pruning

Alpha-beta Pruning is an optional sub-process within the minimax algorithm. It doesn’t affect the outcome of the algorithm, but affects how quickly the algorithm runs. It works by ‘pruning’ (cutting off branches) in the game tree which do not need to be searched, as there already exists a better move. Its called Alpha-beta pruning due to the addition of two variables, alpha and beta. Alpha is the best value that the maximiser currently can guarantee at that level or above, and similarly Beta is the best value that the minimizer currently can guarantee at that level or below.

A diagram of a diagram

Description automatically generated

Figure 4 – Alpha Beta Pruning Initial Game Tree

First, the maximiser chooses B, and the minimizer selects D in response. At this point, the maximiser chooses 5, ensuring that the minimizer's score remains at 5 or lower. Consequently, the minimizer proceeds to E and evaluates its left node, finding a value of 6. Since 6 is greater than the guaranteed maximum score of 5, the minimizer doesn't need to explore further nodes under E, as it has already secured a minimum score of 5.

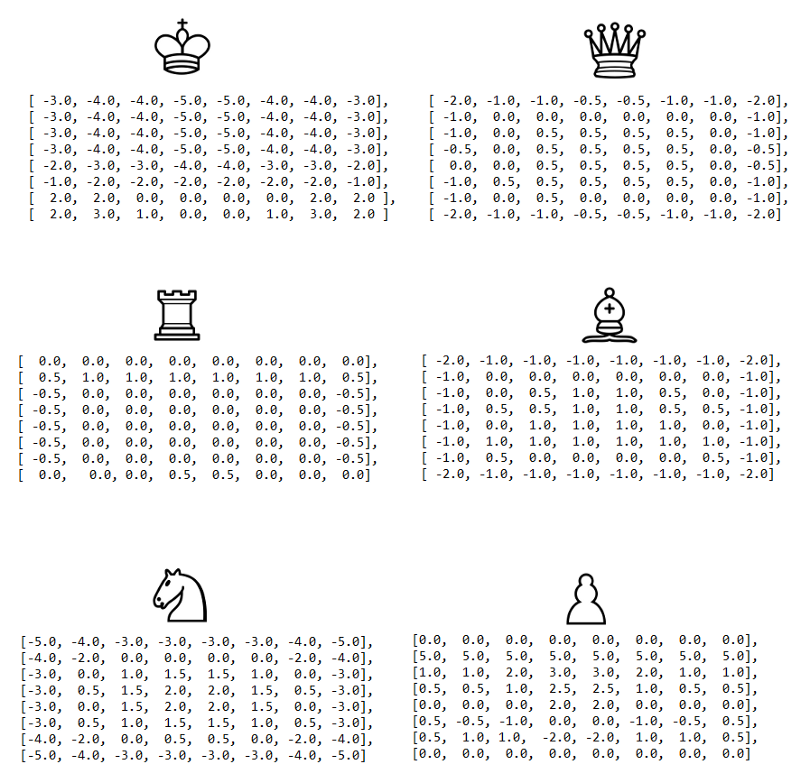
Similarly, upon traversing down to C, the minimizer first selects F. It's again the maximiser’s turn, but upon encountering a value of 1, which doesn't surpass the earlier guaranteed score of 5, it stops exploring this branch. Moving to the right subtree of F, the maximiser discovers a value of 2, which still doesn't exceed the minimum guaranteed score of 5. Recognizing that the minimizer can now ensure a maximum score of 2, which falls short of 5, the maximiser prunes the entire branch stemming from C.

A diagram of a diagram

Description automatically generated

Figure 5 – Final Pruned Game Tree

### Hardcoded Chess Concepts

Previously I explained that the simple evaluation function was not enough to beat the more advanced players. Well, this is due to the limited knowledge the computer is given, it is told that the only way to gain a better position is to capture a piece or to checkmate. However, this is untrue. Positional advantages are also a concept, in chess. Such as you want to develop pieces into the centre of the board. And that the king wants to be protected away into the corner. Now there are many times these concepts need to be broken, such as the endgame. Therefore, we can hardcore some of these concepts into the chess engine, so they favour some moves over others, due to positional advantages.

Here is a piece table, which shows the favoured positions by each piece. This helps hardcode some concepts into the computer, such as developing towards the centre, promotion of pieces, overextending, protecting the king etc.

This paired with the basic evaluation, can lead to a very strong chess engine.

Figure 6 - Chess Piece Table

### GUI

The GUI is the user's gateway to the chess engine, providing an interface that balances simplicity with functionality. While an extravagant design is not the priority, a focus on cleanliness and intuitiveness ensures a seamless user experience.

At the core of the GUI is the interactive chessboard display, where players make their moves. This visual representation of the game allows for easy interaction with the pieces, using methods like drag-and-drop. The chessboard display is designed for responsiveness, ensuring that moves are executed smoothly, contributing to an engaging and visually satisfying gameplay experience.

Alongside the chessboard, the GUI will include a move history feature. This allows users to review the sequence of moves made during the game, allowing post-game analysis and a deeper understanding of the flow of the game. The move history feature aligns with user preferences, enhancing the learning and analytical aspects of the chess-playing experience.

To help with the user interaction, a menu bar is integrated into the GUI, providing quick access to essential functionalities. Users can cycle between different difficulty levels, tailoring the gaming experience to their skill level or desired challenge. The menu bar acts as a centralized command hub, ensuring that users can easily navigate through the engine's features without unnecessary complexity.

### Program Structure

In this section, I will present the program structure of the my project through a series of diagrams. These diagrams offer a visual representation of the relationships and interactions within the projects components.

#### UML Diagram

The UML Diagram provides a visual representation of the structure and relationships within the chess engine project. Despite the project mainly consisting of one main class, the evaluation class, the UML diagram helps show its attributes, methods and interactions. Making it easier to start coding later on.

A computer screen shot of a computer

Description automatically generated

Figure 7 – UML Diagram

A diagram of a software system

Description automatically generated

This diagram above showcases the relationships within the evaluate function, as well as the recursive nation of the minimax algorithm.

Figure 7 – Evaluate Relationship Diagram

Figure 7 – Evaluate Relationship Diagram

The drawing of the chessboard is a simple feature which doesn’t require complex algorithms, however it is still necessary to think about how it will be done before coding.

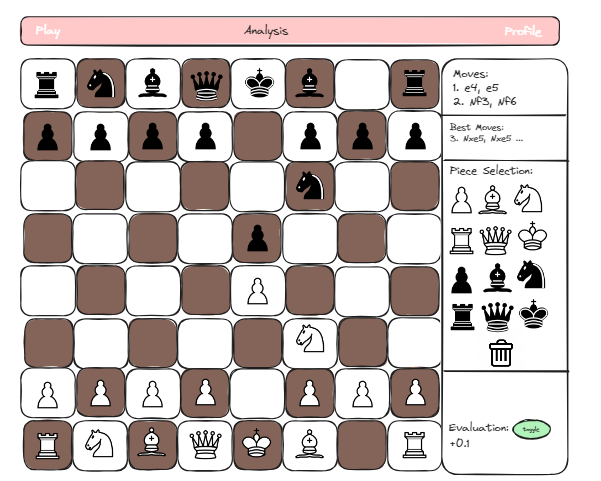
A diagram of a process

Description automatically generated

Figure 8 – Flowchart of drawboard() function

### GUI Design

This subtopic does not need much explanation, this is the basic mock-up design on the GUI: the Play section, the Analysis Section and the Profile Section.

A screenshot of a game

Description automatically generated

Figure 9 – Play Section of the GUI

Figure 10 – Analysis Section of the GUI

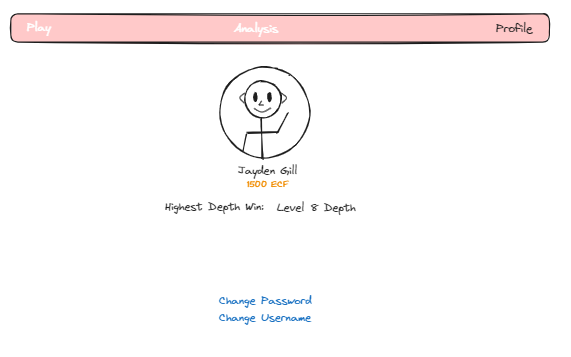


Figure 11 – Profile Section of the GUI

# References

Champion, A. (n.d.). *Stockfish in Depth*. Retrieved from https://towardsdatascience.com/dissecting-stockfish-part-1-in-depth-look-at-a-chess-engine-7fddd1d83579

Champion, A. (n.d.). *Top 5 Chess Engines*. Retrieved from https://royalchessmall.com/en-gb/blogs/blog/5-best-chess-engines

L, A. (n.d.). *Minimax*. Retrieved from Geeks For Geeks: https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-1-introduction/

python-chess. (n.d.). *python-chess*. Retrieved from https://python-chess.readthedocs.io/en/latest/