

NEA Project

Jayden Gill

Chess Engine

UTC Reading

NEA – Chess Engine

Contents

[Section 1: Analysis 2](#_Toc162342515)

[Project Description 2](#_Toc162342516)

[Background and Research 2](#_Toc162342517)

[Stockfish 2](#_Toc162342518)

[End User Interview 2](#_Toc162342519)

[What will I Do? 3](#_Toc162342520)

[Section 2: Documented Design 4](#_Toc162342521)

[System Overview 4](#_Toc162342522)

[Chess Architecture 4](#_Toc162342523)

[Candidate Moves 4](#_Toc162342524)

[Basic-Evaluation Function 5](#_Toc162342525)

[Minimax Algorithm 6](#_Toc162342526)

[Hardcoded Chess Concepts 8](#_Toc162342527)

[GUI 9](#_Toc162342528)

[Program Structure 9](#_Toc162342529)

[GUI Design 11](#_Toc162342530)

[Section 3: Technical Solution 12](#_Toc162342531)

[References 12](#_Toc162342532)

# 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.

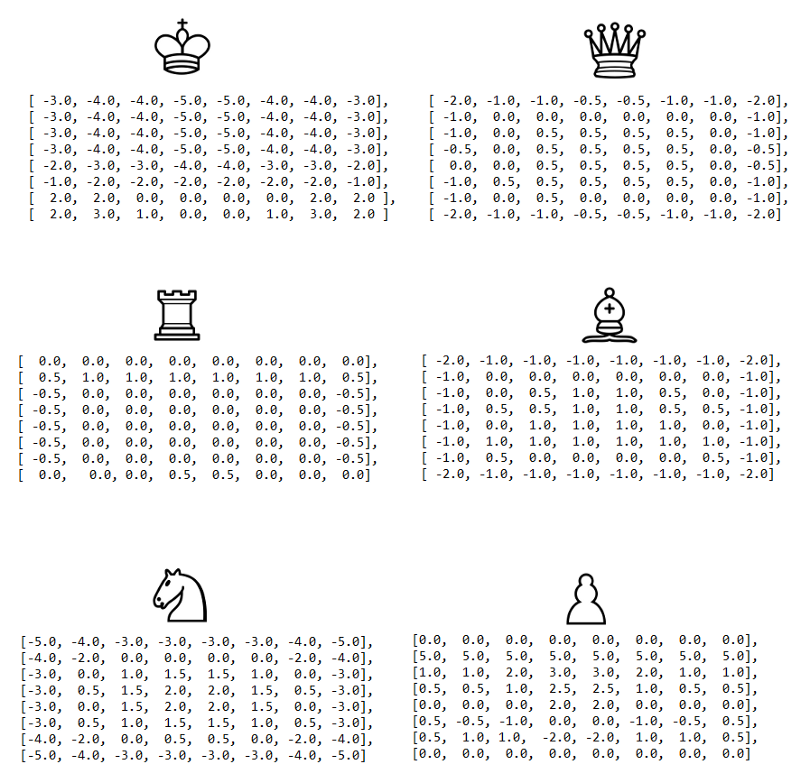
Similarly, upon traversing down to C, the minimizer first selects F. It's again the maximiser’s turn, but upon encountering a value of 1, which doesn't surpass the earlier guaranteed score of 5, it stops exploring this branch. Moving to the right subtree of F, the maximiser discovers a value of 2, which 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.

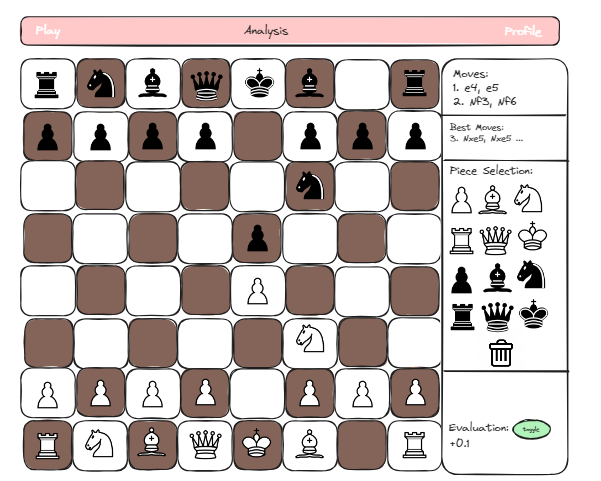
A diagram of a process

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

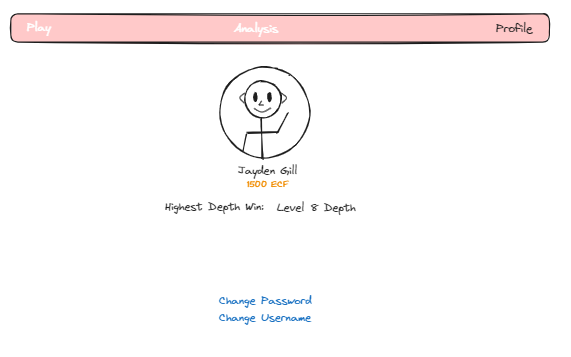


Figure 11 – Profile Section of the GUI

# Section 3: Technical Solution

### chessGui.py Version 1

This was the initial version of the GUI, including draggable pieces, a reference to a future minimax algorithm, and a legal moves checker. Not in classes or functions yet, as this was not yet necessary.

#imports

import pygame

import chess

#setting up the screen

WIDTH, HEIGHT = 480, 480

SQUARE\_SIZE = 60

PIECE\_SIZE = 60

FPS = 60

pygame.init()

screen = pygame.display.set\_mode((WIDTH, HEIGHT))

pygame.display.set\_caption('Chess')

# Piece images

piece\_images = {

    chess.Piece.from\_symbol('P'): pygame.transform.scale(pygame.image.load('assets/whitePawn.png'), (PIECE\_SIZE, PIECE\_SIZE)),

    chess.Piece.from\_symbol('p'): pygame.transform.scale(pygame.image.load('assets/blackPawn.png'), (PIECE\_SIZE, PIECE\_SIZE)),

    chess.Piece.from\_symbol('R'): pygame.transform.scale(pygame.image.load('assets/whiteRook.png'), (PIECE\_SIZE, PIECE\_SIZE)),

    chess.Piece.from\_symbol('r'): pygame.transform.scale(pygame.image.load('assets/blackRook.png'), (PIECE\_SIZE, PIECE\_SIZE)),

    chess.Piece.from\_symbol('N'): pygame.transform.scale(pygame.image.load('assets/whiteKnight.png'), (PIECE\_SIZE, PIECE\_SIZE)),

    chess.Piece.from\_symbol('n'): pygame.transform.scale(pygame.image.load('assets/blackKnight.png'), (PIECE\_SIZE, PIECE\_SIZE)),

    chess.Piece.from\_symbol('B'): pygame.transform.scale(pygame.image.load('assets/whiteBishop.png'), (PIECE\_SIZE, PIECE\_SIZE)),

    chess.Piece.from\_symbol('b'): pygame.transform.scale(pygame.image.load('assets/blackBishop.png'), (PIECE\_SIZE, PIECE\_SIZE)),

    chess.Piece.from\_symbol('Q'): pygame.transform.scale(pygame.image.load('assets/whiteQueen.png'), (PIECE\_SIZE, PIECE\_SIZE)),

    chess.Piece.from\_symbol('q'): pygame.transform.scale(pygame.image.load('assets/blackQueen.png'), (PIECE\_SIZE, PIECE\_SIZE)),

    chess.Piece.from\_symbol('K'): pygame.transform.scale(pygame.image.load('assets/whiteKing.png'), (PIECE\_SIZE, PIECE\_SIZE)),

    chess.Piece.from\_symbol('k'): pygame.transform.scale(pygame.image.load('assets/blackKing.png'), (PIECE\_SIZE, PIECE\_SIZE)),

}

# initializing the board

board = chess.Board()

# Dragging mechanism

dragging = False

dragged\_piece = None

start\_square = None

def draw\_board():

    for i in range(8):

        for j in range(8):  # 8x8 grid

            color = "white" if (i + j) % 2 == 0 else "brown" # alternate colors

            pygame.draw.rect(screen, color, (i \* SQUARE\_SIZE, j \* SQUARE\_SIZE, SQUARE\_SIZE, SQUARE\_SIZE))

    draw\_pieces()

def draw\_pieces():

    for i in range(8):

        for j in range(8):  # 8x8 grid

            piece = board.piece\_at(i + j \* 8)       #converting text to images

            if piece is not None:

                x = i \* SQUARE\_SIZE + (SQUARE\_SIZE - PIECE\_SIZE) // 2

                y = (7 - j) \* SQUARE\_SIZE + (SQUARE\_SIZE - PIECE\_SIZE) // 2

                piece\_image = piece\_images.get(piece)

                if piece\_image is not None:

                    piece\_image = pygame.transform.scale(piece\_image, (PIECE\_SIZE, PIECE\_SIZE))

                    piece\_image\_rect = piece\_image.get\_rect(center=(x + 30, y + 30))

                    screen.blit(piece\_image, piece\_image\_rect)

run = True

timer = pygame.time.Clock()

while run:  # main loop of the game

    timer.tick(FPS)

    screen.fill("lightgray")

    draw\_board()

    for event in pygame.event.get():    #event loop

        if event.type == pygame.QUIT:

            run = False

        elif dragging or (event.type == pygame.MOUSEBUTTONDOWN and event.button == 1):  # mouse click

            if event.type == pygame.MOUSEBUTTONDOWN and event.button == 1:

                x\_coord = event.pos[0] // SQUARE\_SIZE

                y\_coord = 7 - event.pos[1] // SQUARE\_SIZE

                start\_square = chess.square(x\_coord, y\_coord)

                piece = board.piece\_at(start\_square)

                if piece is not None:

                    dragging = True

                    dragged\_piece = piece\_images.get(piece)

                    dragged\_piece = pygame.transform.scale(dragged\_piece, (PIECE\_SIZE, PIECE\_SIZE))

            elif event.type == pygame.MOUSEMOTION and dragging: # mouse motion

                x, y = event.pos

                x -= PIECE\_SIZE // 2

                y -= PIECE\_SIZE // 2

                screen.fill("lightgray")

                draw\_board()

                screen.blit(dragged\_piece, (x, y))

                pygame.display.flip()

            elif event.type == pygame.MOUSEBUTTONUP and event.button == 1 and dragging: # mouse release

                dragging = False

                x\_coord = event.pos[0] // SQUARE\_SIZE

                y\_coord = 7 - event.pos[1] // SQUARE\_SIZE

                end\_square = chess.square(x\_coord, y\_coord)

                move = chess.Move(start\_square, end\_square)

                san\_move = event.dict.get('text') or board.san(move)

                try:

                    move = chess.Move.from\_uci(san\_move)

                except:

                    pass

                if move in board.legal\_moves:

                    board.push(move)

                screen.fill("lightgray")

                draw\_board()

                pygame.display.flip()

                if board.turn == chess.BLACK and not board.is\_game\_over():

                    bestValue, bestMove = minimax(board, 3) # minimax algorithm not yet made

                    board.push\_san(bestMove) # push the best move to the board

                if board.is\_game\_over():

                    fen = board.board\_fen()

                    print(fen) # print the final board state

    pygame.display.flip()

pygame.quit()

### chessPyt.py Version 1

This is the first draft of the minimax algorithm, it uses the basic evaluation to find the best moves resulting in a gain of material. It does not yet take into account, positional advantages.

import chess

board=chess.Board()

def legalMoves(board):  # Returns a list of legal moves in SAN format

    moves = [board.san(move) for move in board.legal\_moves]

    return moves

def basicEval(board): # Returns a basic evaluation of the board

    white = board.occupied\_co[chess.WHITE]

    black = board.occupied\_co[chess.BLACK]

    if board.turn == chess.WHITE and board.is\_checkmate():

        return -1000

    elif board.turn == chess.BLACK and board.is\_checkmate():

        return 1000

    elif board.is\_stalemate() or board.is\_fivefold\_repetition() or board.is\_insufficient\_material():

        return 0

    else:

        return (

            chess.popcount(white & board.pawns) - chess.popcount(black & board.pawns) +     # static evaluation

            3 \* (chess.popcount(white & board.knights) - chess.popcount(black & board.knights)) +

            3 \* (chess.popcount(white & board.bishops) - chess.popcount(black & board.bishops)) +

            5 \* (chess.popcount(white & board.rooks) - chess.popcount(black & board.rooks)) +

            9 \* (chess.popcount(white & board.queens) - chess.popcount(black & board.queens))

        )

def minimax(board, depth):

    if board.is\_game\_over() or depth == 0:  # Base case

        return basicEval(board), None

    if board.turn == chess.WHITE:   # Maximizing player

        best = -10000

        best\_move = None

        for move in legalMoves(board): # Recursively call minimax

            board.push\_san(move)

            val, \_ = minimax(board, depth - 1)

            board.pop()

            if val > best: # Update the best move and its value

                best = val

                best\_move = move

        return best, best\_move # Return the best move and its value

    else:  # Minimizing player

        best = 10000

        best\_move = None

        for move in legalMoves(board): # Recursively call minimax

            board.push\_san(move)

            val, \_ = minimax(board, depth - 1)

            board.pop()

            if val < best: # Update the best move and its value

                best = val

                best\_move = move

        return best, best\_move # Return the best move and its value

## chessPyt.py Version 2

After initial testing, the minimax algorithm was very slow even at low depths. Therefore I decided to add alpha beta pruning now, so testing could be done quicker.

import chess

def legalMoves(board): # Returns a list of legal moves

    moves = [board.san(move) for move in board.legal\_moves]

    return moves

def basicEval(board): # Returns a basic evaluation of the board

    white = board.occupied\_co[chess.WHITE] # Returns the squares occupied

    black = board.occupied\_co[chess.BLACK]

    if board.turn == chess.WHITE and board.is\_checkmate():

        return -1000

    elif board.turn == chess.BLACK and board.is\_checkmate():

        return 1000

    elif board.is\_stalemate() or board.is\_fivefold\_repetition() or board.is\_insufficient\_material():

        return 0

    else:

        return (

            chess.popcount(white & board.pawns) - chess.popcount(black & board.pawns) +

            3 \* (chess.popcount(white & board.knights) - chess.popcount(black & board.knights)) +

            3 \* (chess.popcount(white & board.bishops) - chess.popcount(black & board.bishops)) +

            5 \* (chess.popcount(white & board.rooks) - chess.popcount(black & board.rooks)) +

            9 \* (chess.popcount(white & board.queens) - chess.popcount(black & board.queens))

        )

def minimax(board, depth, alpha=-float('inf'), beta=float('inf')): # minimax algorithm with alpha-beta pruning

    if board.is\_game\_over() or depth == 0:

        return basicEval(board), None

    if board.turn == chess.WHITE: # Maximizing player

        best = -float('inf') # Negative infinity rather than -10000

        best\_move = None

        for move in legalMoves(board):

            board.push\_san(move)

            val, \_ = minimax(board, depth - 1, alpha, beta) # Recursive call with new alpha and beta values

            board.pop()

            if val > best:

                best = val

                best\_move = move

            alpha = max(alpha, best) # Update alpha

            if beta <= alpha: # Pruning

                break

        return best, best\_move # Return the best value and the best move

    else:

        best = float('inf') # Positive infinity rather than 10000

        best\_move = None

        for move in legalMoves(board):

            board.push\_san(move)

            val, \_ = minimax(board, depth - 1, alpha, beta) # Recursive call with new alpha and beta values

            board.pop()

            if val < best:

                best = val

                best\_move = move

            beta = min(beta, best) # Update beta

            if beta <= alpha: # Pruning

                break

        return best, best\_move

### chessPyt.py Version 3

I was planning to access the minmax algorithm from the chessGUI.py file, and it was easier to access from within a class. Therefore I tweaked the code into functions and a class for easier access and better structure.

import chess

board=chess.Board()

class Evaluate():  # added to a class so i can access from different file, easier

    def \_\_init\_\_(self): #initializes the board and move count

        self.board=chess.Board()

        self.moveCount=0

        self.moveList=[]

    def legalMoves(board): #returns a list of legal moves

        moves = [board.san(move) for move in board.legal\_moves]

        return moves

    def basicEval(board): #evaluates the board based on the pieces

        white = board.occupied\_co[chess.WHITE]

        black = board.occupied\_co[chess.BLACK]

        if board.turn == chess.WHITE and board.is\_checkmate():

            return -1000

        elif board.turn == chess.BLACK and board.is\_checkmate():

            return 1000

        elif board.is\_stalemate() or board.is\_fivefold\_repetition() or board.is\_insufficient\_material():

            return 0

        else:

            return (

                chess.popcount(white & board.pawns) - chess.popcount(black & board.pawns) +

                3 \* (chess.popcount(white & board.knights) - chess.popcount(black & board.knights)) +

                3 \* (chess.popcount(white & board.bishops) - chess.popcount(black & board.bishops)) +

                5 \* (chess.popcount(white & board.rooks) - chess.popcount(black & board.rooks)) +

                9 \* (chess.popcount(white & board.queens) - chess.popcount(black & board.queens))

            )

    def minimax(board, depth, alpha=-float('inf'), beta=float('inf')): #minimax algorithm to find the best move

        if board.is\_game\_over() or depth == 0:

            return Evaluate.basicEval(board), None

        if board.turn == chess.WHITE:

            best = -float('inf')

            best\_move = None

            for move in Evaluate.legalMoves(board): #iterates through all legal moves

                board.push\_san(move)

                val, \_ = Evaluate.minimax(board, depth - 1, alpha, beta) #recursively calls minimax

                board.pop()

                if val > best:

                    best = val

                    best\_move = move

                alpha = max(alpha, best) # new alpha value

                if beta <= alpha: #alpha beta pruning

                    break

            return best, best\_move

        else:

            best = float('inf')

            best\_move = None

            for move in Evaluate.legalMoves(board): #iterates through all legal moves

                board.push\_san(move)

                val, \_ = Evaluate.minimax(board, depth - 1, alpha, beta)

                board.pop()

                if val < best:

                    best = val

                    best\_move = move

                beta = min(beta, best) #new beta value

                if beta <= alpha: #alpha beta pruning

                    break

            return best, best\_move

### chessLogin.py Version 1

As I am having users, I needed a login system to access the actual chess engine. Therefore a preliminary login system was needed. I used hashing to make sure passwords were secured stored, so data was protected.

# imports

import tkinter as tk

from tkinter import messagebox

import mysql.connector

from mysql.connector import Error

import bcrypt

def create\_connection():   # function to create connection

    """Create a connection to MySQL database."""

    try:

        connection = mysql.connector.connect(

            host='localhost',

            database='chess\_login',

            user='root',

            password='REDACTED'

        )

        if connection.is\_connected():

            print('Connected to MySQL database')

            return connection

    except Error as e:

        print(f"Error connecting to MySQL database: {e}")

        return None

def close\_connection(connection):   # function to close connection

    """Close the connection to MySQL database."""

    if connection:

        connection.close()

def register\_user():    # function to register user

    """Register a new user."""

    username = username\_entry.get()

    password = password\_entry.get()

    try:

        connection = create\_connection()

        if connection:

            cursor = connection.cursor()

            hashed\_password = bcrypt.hashpw(password.encode('utf-8'), bcrypt.gensalt()) # hash password

            query = "INSERT INTO users (username, password) VALUES (%s, %s)" # query to insert data into table

            cursor.execute(query, (username, hashed\_password))

            connection.commit()

            messagebox.showinfo("Success", "User registered successfully!")

            close\_connection(connection)

    except Error as e:

        messagebox.showerror("Error", f"Error registering user: {e}")

        close\_connection(connection)

def login\_user():  # function to login user

    """Authenticate user login."""

    username = username\_entry.get()

    password = password\_entry.get()

    try:

        connection = create\_connection()

        if connection:

            cursor = connection.cursor()

            query = "SELECT password FROM users WHERE username = %s"

            cursor.execute(query, (username,))

            user\_data = cursor.fetchone()

            if user\_data:

                stored\_password = user\_data[0].encode('utf-8')

                if bcrypt.checkpw(password.encode('utf-8'), stored\_password): # check if password is correct

                    messagebox.showinfo("Success", "Login successful!")

                else:

                    messagebox.showerror("Error", "Invalid username or password.")

            else:

                messagebox.showerror("Error", "User not found.")

            close\_connection(connection)

    except Error as e:

        messagebox.showerror("Error", f"Error logging in: {e}")

        close\_connection(connection)

root = tk.Tk()

root.title("User Authentication")

# Username label and entry

username\_label = tk.Label(root, text="Username:")

username\_label.grid(row=0, column=0, padx=10, pady=5, sticky="e")

username\_entry = tk.Entry(root)

username\_entry.grid(row=0, column=1, padx=10, pady=5)

# Password label and entry

password\_label = tk.Label(root, text="Password:")

password\_label.grid(row=1, column=0, padx=10, pady=5, sticky="e")

password\_entry = tk.Entry(root, show="\*")

password\_entry.grid(row=1, column=1, padx=10, pady=5)

# Register and login buttons

register\_button = tk.Button(root, text="Register", command=register\_user)

register\_button.grid(row=2, column=0, columnspan=2, pady=10)

login\_button = tk.Button(root, text="Login", command=login\_user)

login\_button.grid(row=3, column=0, columnspan=2, pady=5)

root.mainloop()

### chessPyt.py Version 4

We now had a basic evaluation system, making a pretty basic engine; which would beat lesser players. However it would struggle against more advanced players, therefore a better evaluation system was needed. Using piece tables, to ‘hardcode’ chess concepts into the engine.

import chess

board=chess.Board()

class Evaluate():

    def legalMoves(board): # returns the legal moves of the board

        moves = [board.san(move) for move in board.legal\_moves]

        return moves

    def basicEval(board): # returns the evaluation of the board based on the material count

        white = board.occupied\_co[chess.WHITE]

        black = board.occupied\_co[chess.BLACK]

        if board.turn == chess.WHITE and board.is\_checkmate():

            return -1000

        elif board.turn == chess.BLACK and board.is\_checkmate():

            return 1000

        elif board.is\_stalemate() or board.is\_fivefold\_repetition() or board.is\_insufficient\_material():

            return 0

        else:

            return (

                chess.popcount(white & board.pawns) - chess.popcount(black & board.pawns) +

                3 \* (chess.popcount(white & board.knights) - chess.popcount(black & board.knights)) +

                3 \* (chess.popcount(white & board.bishops) - chess.popcount(black & board.bishops)) +

                5 \* (chess.popcount(white & board.rooks) - chess.popcount(black & board.rooks)) +

                9 \* (chess.popcount(white & board.queens) - chess.popcount(black & board.queens))

            )

    def minimax(board, depth, PST, alpha=-float('inf'), beta=float('inf')): # returns the best move based on the minimax algorithm

        if board.is\_game\_over() or depth == 0:

            return Evaluate.overallEval(board,PST), None

        if board.turn == chess.WHITE: # maximizing player

            best = -float('inf')

            best\_move = None

            for move in Evaluate.legalMoves(board):

                board.push\_san(move)

                val, \_ = Evaluate.minimax(board, depth - 1,PST, alpha, beta)

                board.pop()

                if val > best:

                    best = val

                    best\_move = move

                alpha = max(alpha, best)

                if beta <= alpha:

                    break

            return best, best\_move

        else: # minimizing player

            best = float('inf')

            best\_move = None

            for move in Evaluate.legalMoves(board):

                board.push\_san(move)

                val, \_ = Evaluate.minimax(board, depth - 1, PST, alpha, beta)

                board.pop()

                if val < best:

                    best = val

                    best\_move = move

                beta = min(beta, best)

                if beta <= alpha:

                    break

            return best, best\_move

    def pTableEval(board, piece, square, PST, isWhite): # returns the evaluation of the piece based on the piece square tables

        if str(piece).lower() == 'p' :

            if isWhite:

                return PST[chess.PAWN][square]

            else:

                return -PST[chess.PAWN][chess.square\_mirror(square)]

        elif str(piece).lower() == 'n':

            if isWhite:

                return PST[chess.KNIGHT][square]

            else:

                return -PST[chess.KNIGHT][chess.square\_mirror(square)]

        elif str(piece).lower() == 'b':

            if isWhite:

                return PST[chess.BISHOP][square]

            else:

                return -PST[chess.BISHOP][chess.square\_mirror(square)]

        elif str(piece).lower() == 'r':

            if isWhite:

                return PST[chess.ROOK][square]

            else:

                return -PST[chess.ROOK][chess.square\_mirror(square)]

        elif str(piece).lower() == 'q':

            if isWhite:

                return PST[chess.QUEEN][square]

            else:

                return -PST[chess.QUEEN][chess.square\_mirror(square)]

        elif str(piece).lower() == 'k':

            if isWhite:

                return PST[chess.KING][square]

            else:

                return -PST[chess.KING][chess.square\_mirror(square)]

        else:

            return 0

    def symbol(piece): # returns the symbol of the piece

        if piece == 1:

            return "p"

        elif piece == 2:

            return "n"

        elif piece == 3:

            return "b"

        elif piece == 4:

            return "r"

        elif piece == 5:

            return "q"

        elif piece == 6:

            return "k"

    def pTableScore(board, PST): # returns the evaluation of the board based on the piece square tables

        score = 0

        for i in range(64):

            piece = board.piece\_at(i)

            if str(piece) == 'None':

                pass

            elif str(piece).isupper():

                isWhite = True

            else:

                isWhite = False

            if piece is not None:

                score += Evaluate.pTableEval(board, piece.symbol(), i, PST, isWhite)

        return score

    def overallEval(board,PST): # returns the evaluation of the board

        return float(Evaluate.basicEval(board)) + float(Evaluate.pTableScore(board,PST))

# individual piece square tables

pawn\_table = [

    0, 0, 0, 0, 0, 0, 0, 0,

    5, 5, 5, 5, 5, 5 ,5 ,5,

    1, 1, 2, 3, 3, 2, 1, 1,

    0.5, 0.5, 1, 2.5, 2.5, 1, 0.5, 0.5,

    0, 0, 0, 2, 2, 0, 0, 0,

    0.5, -0.5, -1, 0, 0, -1, -0.5, 0.5,

    0.5, 1, 1, -2, -2, 1, 1, 0.5,

    0, 0, 0, 0, 0, 0, 0, 0

]

knights\_table = [

    -5, -4, -3, -3, -3, -3, -4, -5,

    -4, -2, 0, 0, 0, 0, -2, -4,

    -3, 0, 1, 1.5, 1.5, 1, 0, -3,

    -3, 0.5, 1.5, 2, 2, 1.5, 0.5, -3,

    -3, 0, 1.5, 2, 2, 1.5, 0, -3,

    -3, 0.5, 1, 1.5, 1.5, 1, 0.5, -3,

    -4, -2, 0, 0.5, 0.5, 0, -2, -4,

    -5, -4, -3, -3, -3, -3, -4, -5

]

bishops\_table = [

    -2, -1, -1, -1, -1, -1, -1, -2,

    -1, 0, 0, 0, 0, 0, 0, -1,

    -1, 0, 0.5, 1, 1, 0.5, 0, -1,

    -1, 0.5, 0.5, 1, 1, 0.5, 0.5, -1,

    -1, 0, 1, 1, 1, 1, 0, -1,

    -1, 1, 1, 1, 1, 1, 1, -1,

    -1, 0.5, 0, 0, 0, 0, 0.5, -1,

    -2, -1, -1, -1, -1, -1, -1, -2

]

rooks\_table = [

    0, 0, 0, 0, 0, 0, 0, 0,

    0.5, 1, 1, 1, 1, 1, 1, 0.5,

    -0.5, 0, 0, 0, 0, 0, 0, -0.5,

    -0.5, 0, 0, 0, 0, 0, 0, -0.5,

    -0.5, 0, 0, 0, 0, 0, 0, -0.5,

    -0.5, 0, 0, 0, 0, 0, 0, -0.5,

    -0.5, 0, 0, 0, 0, 0, 0, -0.5,

    0, 0, 0, 0.5, 0.5, 0, 0, 0

]

queens\_table = [

    -2, -1, -1, -0.5, -0.5, -1, -1, -2,

    -1, 0, 0, 0, 0, 0, 0, -1,

    -1, 0, 0.5, 0.5, 0.5, 0.5, 0, -1,

    -0.5, 0, 0.5, 0.5, 0.5, 0.5, 0, -0.5,

    0, 0, 0.5, 0.5, 0.5, 0.5, 0, -0.5,

    -1, 0.5, 0.5, 0.5, 0.5, 0.5, 0, -1,

    -1, 0, 0.5, 0, 0, 0, 0, -1,

    -2, -1, -1, -0.5, -0.5, -1, -1, -2

]

kings\_table = [

    -3, -4, -4, -5, -5, -4, -4, -3,

    -3, -4, -4, -5, -5, -4, -4, -3,

    -3, -4, -4, -5, -5, -4, -4, -3,

    -3, -4, -4, -5, -5, -4, -4, -3,

    -2, -3, -3, -4, -4, -3, -3, -2,

    -1, -2, -2, -2, -2, -2, -2, -1,

    2, 2, 0, 0, 0, 0, 2, 2,

    2, 3, 1, 0, 0, 1, 3, 2

]

# piece square tables

PST = {

    chess.PAWN: list(reversed(pawn\_table)),

    chess.KNIGHT: list(reversed(knights\_table)),

    chess.BISHOP: list(reversed(bishops\_table)),

    chess.ROOK: list(reversed(rooks\_table)),

    chess.QUEEN: list(reversed(queens\_table)),

    chess.KING: list(reversed(kings\_table))

}

### chessGui.py Version 2

As I was testing, I noticed that the player could not promote any pieces as I had not handled this event graphically. Therefore I needed a way to not only promote a piece, but ask the user what piece they want to promote to. I also added in the evaluation.minimax() function into the program. I also noticed that the AI was very quick once down to little amount of pieces, this was due to it having to do less computation. Therefore, when a small amount of pieces were left, it would increase the depth, making it make more informed decisions.

# Importing the required libraries

import pygame

import chess

import chessPyt

import promoPopUp

WIDTH, HEIGHT = 480, 480

SQUARE\_SIZE = 60

PIECE\_SIZE = 60

FPS = 60

pygame.init()

screen = pygame.display.set\_mode((WIDTH, HEIGHT))

pygame.display.set\_caption('Chess')

piece\_images = {

    chess.Piece.from\_symbol('P'): pygame.transform.scale(pygame.image.load('assets/whitePawn.png'), (PIECE\_SIZE, PIECE\_SIZE)),

    chess.Piece.from\_symbol('p'): pygame.transform.scale(pygame.image.load('assets/blackPawn.png'), (PIECE\_SIZE, PIECE\_SIZE)),

    chess.Piece.from\_symbol('R'): pygame.transform.scale(pygame.image.load('assets/whiteRook.png'), (PIECE\_SIZE, PIECE\_SIZE)),

    chess.Piece.from\_symbol('r'): pygame.transform.scale(pygame.image.load('assets/blackRook.png'), (PIECE\_SIZE, PIECE\_SIZE)),

    chess.Piece.from\_symbol('N'): pygame.transform.scale(pygame.image.load('assets/whiteKnight.png'), (PIECE\_SIZE, PIECE\_SIZE)),

    chess.Piece.from\_symbol('n'): pygame.transform.scale(pygame.image.load('assets/blackKnight.png'), (PIECE\_SIZE, PIECE\_SIZE)),

    chess.Piece.from\_symbol('B'): pygame.transform.scale(pygame.image.load('assets/whiteBishop.png'), (PIECE\_SIZE, PIECE\_SIZE)),

    chess.Piece.from\_symbol('b'): pygame.transform.scale(pygame.image.load('assets/blackBishop.png'), (PIECE\_SIZE, PIECE\_SIZE)),

    chess.Piece.from\_symbol('Q'): pygame.transform.scale(pygame.image.load('assets/whiteQueen.png'), (PIECE\_SIZE, PIECE\_SIZE)),

    chess.Piece.from\_symbol('q'): pygame.transform.scale(pygame.image.load('assets/blackQueen.png'), (PIECE\_SIZE, PIECE\_SIZE)),

    chess.Piece.from\_symbol('K'): pygame.transform.scale(pygame.image.load('assets/whiteKing.png'), (PIECE\_SIZE, PIECE\_SIZE)),

    chess.Piece.from\_symbol('k'): pygame.transform.scale(pygame.image.load('assets/blackKing.png'), (PIECE\_SIZE, PIECE\_SIZE)),

}

board = chess.Board()

dragging = False

dragged\_piece = None

start\_square = None

def draw\_board(): # Draw the chess board

    for i in range(8):

        for j in range(8): # 8x8 grid with alternating colors

            color = "white" if (i + j) % 2 == 0 else "brown"

            pygame.draw.rect(screen, color, (i \* SQUARE\_SIZE, j \* SQUARE\_SIZE, SQUARE\_SIZE, SQUARE\_SIZE))

    draw\_pieces()

def draw\_pieces():

    for i in range(8):

        for j in range(8): # Draw the pieces on the board

            piece = board.piece\_at(i + j \* 8) # Get the piece at the given square

            if piece is not None:

                x = i \* SQUARE\_SIZE + (SQUARE\_SIZE - PIECE\_SIZE) // 2

                y = (7 - j) \* SQUARE\_SIZE + (SQUARE\_SIZE - PIECE\_SIZE) // 2

                piece\_image = piece\_images.get(piece)

                if piece\_image is not None:

                    piece\_image = pygame.transform.scale(piece\_image, (PIECE\_SIZE, PIECE\_SIZE))

                    piece\_image\_rect = piece\_image.get\_rect(center=(x + 30, y + 30))

                    screen.blit(piece\_image, piece\_image\_rect)

run = True

timer = pygame.time.Clock()

while run:

    timer.tick(FPS)

    screen.fill("lightgray")

    draw\_board()

    for event in pygame.event.get(): # Check for events

        if event.type == pygame.QUIT:

            run = False

        elif dragging or (event.type == pygame.MOUSEBUTTONDOWN and event.button == 1): # Check if the mouse is clicked

            if event.type == pygame.MOUSEBUTTONDOWN and event.button == 1:

                x\_coord = event.pos[0] // SQUARE\_SIZE

                y\_coord = 7 - event.pos[1] // SQUARE\_SIZE

                start\_square = chess.square(x\_coord, y\_coord)

                piece = board.piece\_at(start\_square)

                if piece is not None:

                    dragging = True

                    dragged\_piece = piece\_images.get(piece)

                    dragged\_piece = pygame.transform.scale(dragged\_piece, (PIECE\_SIZE, PIECE\_SIZE))

            elif event.type == pygame.MOUSEMOTION and dragging: # Check if the mouse is moved

                x, y = event.pos

                x -= PIECE\_SIZE // 2

                y -= PIECE\_SIZE // 2

                screen.fill("lightgray")

                draw\_board()

                screen.blit(dragged\_piece, (x, y))

                pygame.display.flip()

            elif event.type == pygame.MOUSEBUTTONUP and event.button == 1 and dragging: # Check if the mouse is released

                dragging = False

                x\_coord = event.pos[0] // SQUARE\_SIZE

                y\_coord = 7 - event.pos[1] // SQUARE\_SIZE

                end\_square = chess.square(x\_coord, y\_coord)

                move = chess.Move(start\_square, end\_square)

                if board.piece\_at(start\_square).piece\_type == chess.PAWN and chess.square\_rank(end\_square) in (0, 7): # Check if the pawn reaches the end of the board

                    promotionMoves = []

                    for x in board.legal\_moves:

                        if str(x)[-1] in ["q", "r", "b", "n"]: # Check if the move is a promotion move

                            promotionMoves.append(x) # Add the move to the list of promotion moves

                    if promotionMoves != []: # Check if there are any promotion moves

                        promoPiece = promoPopUp.create\_popup\_window() # Create a popup window to select the promotion piece

                        for x in promotionMoves: # Check if the move is a promotion move

                            if str(x)[-1] == promoPiece:

                                move = x

                                break

                san\_move = event.dict.get('text') or board.san(move)

                try:

                    move = chess.Move.from\_uci(san\_move)

                except:

                    pass

                if move in board.legal\_moves:

                    board.push(move)

                screen.fill("lightgray")

                draw\_board()

                pygame.display.flip()

                if board.turn == chess.BLACK and not board.is\_game\_over():

                    numLeft = chessPyt.numPiecesLeft(board)

                    if numLeft < 15: #improves depth if low amount of pieces

                        depth = 4

                    elif numLeft < 10:

                        depth = 5

                    else:

                        depth = 3

                    bestValue, bestMove = chessPyt.Evaluate.minimax(board, depth, chessPyt.PST) # Get the best move for the AI

                    print("Evaluation: ", bestValue, "Best Move: ", bestMove) # Print the evaluation and the best move

                    board.push\_san(bestMove) # Make the move

                if board.is\_game\_over():

                    fen = board.board\_fen()

                    print(fen)

    pygame.display.flip()

pygame.quit()

### promoPopUp.py Version 1

This is the python file which handles the promotion for the player, it pops up a new window for the player to select the piece and then gives the players option to the chessGUI.py program.

# imports

import pygame

import sys

# Initialize Pygame

pygame.init()

# Constants

WINDOW\_WIDTH = 400

WINDOW\_HEIGHT = 100

BLACK = (0, 0, 0)

WHITE = (255, 255, 255)

GRAY = (200, 200, 200)

# Function to create a popup window

def create\_popup\_window():

    screen = pygame.display.set\_mode((WINDOW\_WIDTH, WINDOW\_HEIGHT))

    pygame.display.set\_caption("Select Tool")

    # Load images

    image1 = pygame.image.load("assets/whiteQueen.png")

    image2 = pygame.image.load("assets/whiteRook.png")

    image3 = pygame.image.load("assets/whiteBishop.png")

    image4 = pygame.image.load("assets/whiteKnight.png")

    images = [image1, image2, image3, image4]

    # Define button dimensions and positions

    button\_width = 75

    button\_height = 75

    button\_padding = 0

    num\_buttons = len(images)

    total\_width = num\_buttons \* button\_width + (num\_buttons - 1) \* button\_padding

    start\_x = (WINDOW\_WIDTH - total\_width) // 2

    y = (WINDOW\_HEIGHT - button\_height) // 2

    # Draw buttons

    for i, image in enumerate(images):

        x = start\_x + (button\_width + button\_padding) \* i

        button\_rect = pygame.Rect(x, y, button\_width, button\_height)

        screen.blit(image, button\_rect)

    pygame.display.flip()

    # Main loop

    while True:

        for event in pygame.event.get():

            if event.type == pygame.QUIT:

                pygame.quit()

                sys.exit()

            elif event.type == pygame.MOUSEBUTTONDOWN:

                mouse\_pos = pygame.mouse.get\_pos()

                # Check which button is clicked

                for i, image in enumerate(images):

                    x = start\_x + (button\_width + button\_padding) \* i

                    button\_rect = pygame.Rect(x, y, button\_width, button\_height)

                    if button\_rect.collidepoint(mouse\_pos):

                        if i + 1 == 1:

                            return "q"

                        elif i + 1 == 2:

                            return "r"

                        elif i + 1 == 3:

                            return "b"

                        elif i + 1 == 4:

                            return "n"

### chessPyt.py Version 5

The piece tables I had previously implemented were working, the AI would make better moves: controlling the centre, hiding their king away. However, due to the piece table values, being greater than the piece values; it would sacrifice pieces for no reason, other than to gain central space. Which sometimes is necessary, however it was not in these cases. Due to this, I had to divide all the piece table values by 10, and this fixed the sacrifice problem but kept the hardcoded concepts working.

def pTableEval(board, piece, square, PST, isWhite):

        if str(piece).lower() == 'p' :

            if isWhite:

                return PST[chess.PAWN][square] / 10

            else:

                return -PST[chess.PAWN][chess.square\_mirror(square)] / 10

        elif str(piece).lower() == 'n':

            if isWhite:

                return PST[chess.KNIGHT][square] / 10

            else:

                return -PST[chess.KNIGHT][chess.square\_mirror(square)] / 10

        elif str(piece).lower() == 'b':

            if isWhite:

                return PST[chess.BISHOP][square] / 10

            else:

                return -PST[chess.BISHOP][chess.square\_mirror(square)] / 10

        elif str(piece).lower() == 'r':

            if isWhite:

                return PST[chess.ROOK][square] / 10

            else:

                return -PST[chess.ROOK][chess.square\_mirror(square)] / 10

        elif str(piece).lower() == 'q':

            if isWhite:

                return PST[chess.QUEEN][square] / 10

            else:

                return -PST[chess.QUEEN][chess.square\_mirror(square)] / 10

        elif str(piece).lower() == 'k':

            if isWhite:

                return PST[chess.KING][square] / 10

            else:

                return -PST[chess.KING][chess.square\_mirror(square)] / 10

        else:

            return 0

### chessGui.py Version 3

As I previously discussed, I want to be able to see previous moves so that you can see what has happened. Therefore I implemented a sidebar, containing black and whites previous moves.

#imports

import pygame

import chess

import chessPyt

import promoPopUp

WIDTH, HEIGHT = 600, 480  # Adjusted width to accommodate the sidebar

SQUARE\_SIZE = 60

PIECE\_SIZE = 60

FPS = 60

# Sidebar dimensions

SIDEBAR\_WIDTH = 120

MOVE\_HISTORY\_HEIGHT = HEIGHT

pygame.init()

screen = pygame.display.set\_mode((WIDTH, HEIGHT))

pygame.display.set\_caption('Chess')

# Create a font for displaying move history

font = pygame.font.SysFont('Arial', 20)

# Load piece images

piece\_images = {

    chess.Piece.from\_symbol('P'): pygame.transform.scale(pygame.image.load('assets/whitePawn.png'), (PIECE\_SIZE, PIECE\_SIZE)),

    chess.Piece.from\_symbol('p'): pygame.transform.scale(pygame.image.load('assets/blackPawn.png'), (PIECE\_SIZE, PIECE\_SIZE)),

    chess.Piece.from\_symbol('R'): pygame.transform.scale(pygame.image.load('assets/whiteRook.png'), (PIECE\_SIZE, PIECE\_SIZE)),

    chess.Piece.from\_symbol('r'): pygame.transform.scale(pygame.image.load('assets/blackRook.png'), (PIECE\_SIZE, PIECE\_SIZE)),

    chess.Piece.from\_symbol('N'): pygame.transform.scale(pygame.image.load('assets/whiteKnight.png'), (PIECE\_SIZE, PIECE\_SIZE)),

    chess.Piece.from\_symbol('n'): pygame.transform.scale(pygame.image.load('assets/blackKnight.png'), (PIECE\_SIZE, PIECE\_SIZE)),

    chess.Piece.from\_symbol('B'): pygame.transform.scale(pygame.image.load('assets/whiteBishop.png'), (PIECE\_SIZE, PIECE\_SIZE)),

    chess.Piece.from\_symbol('b'): pygame.transform.scale(pygame.image.load('assets/blackBishop.png'), (PIECE\_SIZE, PIECE\_SIZE)),

    chess.Piece.from\_symbol('Q'): pygame.transform.scale(pygame.image.load('assets/whiteQueen.png'), (PIECE\_SIZE, PIECE\_SIZE)),

    chess.Piece.from\_symbol('q'): pygame.transform.scale(pygame.image.load('assets/blackQueen.png'), (PIECE\_SIZE, PIECE\_SIZE)),

    chess.Piece.from\_symbol('K'): pygame.transform.scale(pygame.image.load('assets/whiteKing.png'), (PIECE\_SIZE, PIECE\_SIZE)),

    chess.Piece.from\_symbol('k'): pygame.transform.scale(pygame.image.load('assets/blackKing.png'), (PIECE\_SIZE, PIECE\_SIZE)),

}

board = chess.Board()

# Variables to handle dragging

dragging = False

dragged\_piece = None

start\_square = None

# List to store move history

move\_history = []

def draw\_board():

    for i in range(8):

        for j in range(8): # Draw squares with alternating colors

            color = "white" if (i + j) % 2 == 0 else "brown"

            pygame.draw.rect(screen, color, (i \* SQUARE\_SIZE, j \* SQUARE\_SIZE, SQUARE\_SIZE, SQUARE\_SIZE))

    draw\_pieces()

def draw\_pieces():

    for i in range(8):

        for j in range(8): # Draw pieces on the board

            piece = board.piece\_at(i + j \* 8) # Get piece from board

            if piece is not None:

                x = i \* SQUARE\_SIZE + (SQUARE\_SIZE - PIECE\_SIZE) // 2

                y = (7 - j) \* SQUARE\_SIZE + (SQUARE\_SIZE - PIECE\_SIZE) // 2

                piece\_image = piece\_images.get(piece)

                if piece\_image is not None:

                    piece\_image = pygame.transform.scale(piece\_image, (PIECE\_SIZE, PIECE\_SIZE))

                    piece\_image\_rect = piece\_image.get\_rect(center=(x + 30, y + 30))

                    screen.blit(piece\_image, piece\_image\_rect)

# new draw\_move\_history function

def draw\_move\_history():

    # Create sidebar surface

    sidebar = pygame.Surface((SIDEBAR\_WIDTH, MOVE\_HISTORY\_HEIGHT))

    sidebar.fill((200, 200, 200))

    # Render and blit move history

    y\_offset = 10

    white\_moves = []

    black\_moves = []

    # Separate white and black moves

    for i, move in enumerate(move\_history):

        if i % 2 == 0:

            white\_moves.append(move)

        else:

            black\_moves.append(move)

    i = 0

    # Render and blit moves

    for white\_move, black\_move in zip(white\_moves, black\_moves):

        i += 1

        move\_text = f"{i}.{white\_move} {black\_move}"

        text = font.render(move\_text, True, (0, 0, 0))

        sidebar.blit(text, (10, y\_offset))

        y\_offset += font.get\_linesize()

    screen.blit(sidebar, (WIDTH - SIDEBAR\_WIDTH, 0))

run = True

timer = pygame.time.Clock()

while run:

    timer.tick(FPS)

    screen.fill("lightgray")

    draw\_board()

    # Draw move history

    draw\_move\_history()

    for event in pygame.event.get():

        if event.type == pygame.QUIT:

            run = False

        elif dragging or (event.type == pygame.MOUSEBUTTONDOWN and event.button == 1):

            if event.type == pygame.MOUSEBUTTONDOWN and event.button == 1:

                x\_coord = event.pos[0] // SQUARE\_SIZE

                y\_coord = 7 - event.pos[1] // SQUARE\_SIZE

                start\_square = chess.square(x\_coord, y\_coord)

                piece = board.piece\_at(start\_square)

                if piece is not None:

                    dragging = True

                    dragged\_piece = piece\_images.get(piece)

                    dragged\_piece = pygame.transform.scale(dragged\_piece, (PIECE\_SIZE, PIECE\_SIZE))

            elif event.type == pygame.MOUSEMOTION and dragging:

                x, y = event.pos

                x -= PIECE\_SIZE // 2

                y -= PIECE\_SIZE // 2

                screen.fill("lightgray")

                draw\_board()

                draw\_move\_history()

                screen.blit(dragged\_piece, (x, y))

                pygame.display.flip()

            elif event.type == pygame.MOUSEBUTTONUP and event.button == 1 and dragging:

                dragging = False

                x\_coord = event.pos[0] // SQUARE\_SIZE

                y\_coord = 7 - event.pos[1] // SQUARE\_SIZE

                end\_square = chess.square(x\_coord, y\_coord)

                move = chess.Move(start\_square, end\_square)

                if board.piece\_at(start\_square).piece\_type == chess.PAWN and chess.square\_rank(end\_square) in (0, 7):

                    promotionMoves = []

                    for x in board.legal\_moves:

                        if str(x)[-1] in ["q", "r", "b", "n"]:

                            promotionMoves.append(x)

                    if promotionMoves != []:

                        promoPiece = promoPopUp.create\_popup\_window()

                        screen = pygame.display.set\_mode((WIDTH, HEIGHT))

                        for x in promotionMoves:

                            if str(x)[-1] == promoPiece:

                                move = x

                                break

                san\_move = event.dict.get('text') or board.san(move)

                try:

                    move = chess.Move.from\_uci(san\_move)

                except:

                    pass

                if move in board.legal\_moves:

                    board.push(move)

                    move\_history.append(san\_move)  # Add move to history

                screen.fill("lightgray")

                draw\_board()

                draw\_move\_history()

                pygame.display.flip()

                if board.turn == chess.BLACK and not board.is\_game\_over():

                    numLeft = chessPyt.numPiecesLeft(board)

                    if numLeft < 15:

                        depth = 4

                    elif numLeft < 10:

                        depth = 5

                    else:

                        depth = 3

                    bestValue, bestMove = chessPyt.Evaluate.minimax(board, depth, chessPyt.PST)

                    print("Evaluation : ", bestValue, "Best Move: ", bestMove)

                    board.push\_san(bestMove)

                    move\_history.append(bestMove)

                if board.is\_game\_over():

                    fen = board.board\_fen()

                    print(fen)

    pygame.display.flip()

pygame.quit()

### chessGui.py Version 4

I then realised that I could also display the evaluation in the side bar as well.

def draw\_move\_history():

    sidebar = pygame.Surface((SIDEBAR\_WIDTH, MOVE\_HISTORY\_HEIGHT))

    sidebar.fill((200, 200, 200))

    # Render and blit move history

    y\_offset = 10

    white\_moves = []

    black\_moves = []

    for i, move in enumerate(move\_history):

        if i % 2 == 0:

            white\_moves.append(move)

        else:

            black\_moves.append(move)

    i = 0

    text = font.render("Moves: ", True, (0, 0, 0))

    sidebar.blit(text, (10, y\_offset))

    y\_offset += font.get\_linesize()

    for white\_move, black\_move in zip(white\_moves, black\_moves):

        i += 1

        move\_text = f"{i}.{white\_move} {black\_move}"

        text = font.render(move\_text, True, (0, 0, 0))

        sidebar.blit(text, (10, y\_offset))

        y\_offset += font.get\_linesize()

    # eval text at bottom of sidebar

    text = font.render(f"Eval: {eval}", True, (0, 0, 0))

    sidebar.blit(text, (10, HEIGHT - font.get\_linesize()))  # Placing evaluation at the bottom

# 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/