NEA
Marcus Lee
A Chess Application

Contents

Analysis

Background to the problem

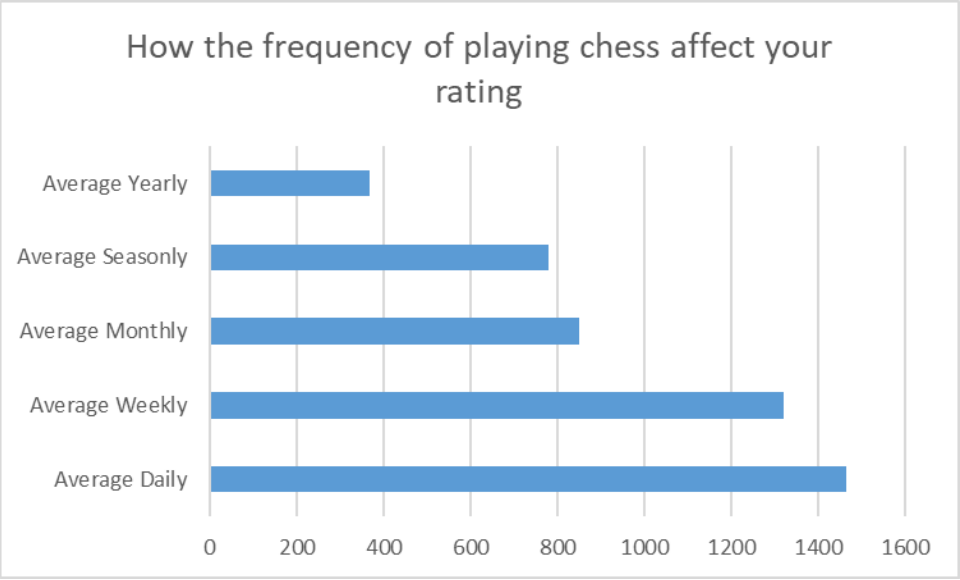
Chess is a logical board game, which contains no element of chance. In Chess, each player must make a calculated decision depending on the moves made by the opponents. I have chosen this problem as within the chess club at my school, I felt there could be more resources available to help develop players' chess skills. Therefore, my goal is to produce an application to help practice for games against other students and so improve their record in competitions.

Research with end users

The end users of my solution would be people from a wide skill range as it would be developed for students from Year 7 to Year 13. Therefore, I have selected my end users appropriately.

My end user is Ethan an avid chess player. I have chosen to interview someone who has experience using the existing solutions as it will allow me to gain valuable insight into important features which he believes are necessary within my solution to improve the user's experience.

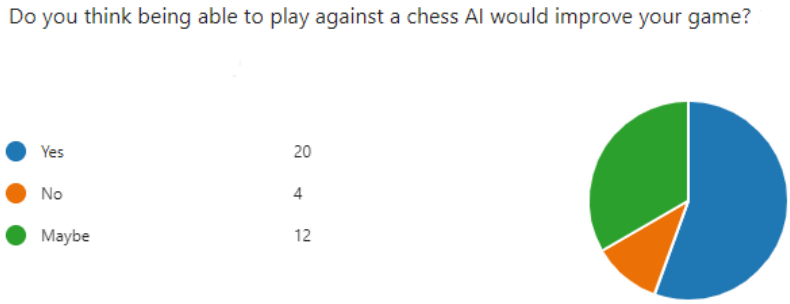
When I first talked to him about programming a chess application, he was thrilled with the idea stating how it would "allow [him] to play when [he] wanted to rather than on a dictated timetable based off [his] local club". He was adamant about the importance of being able to play as frequently as possible as he said the thing that he is currently trying to work on is "getting in as many hours of playing as possible to improve through seeing more and more situations and learning from them". This idea of playing as frequently as possible is reiterated by the survey I carried out on a survey size of 50 people. I then produced the graph below to visual the results:



In this graph, for each group, I calculated a mean rating. So, using this data, it shows a correlation that the more you play the higher your chess rating will be. So, for my app, I aim to make it convenient for users to be able to play and so to improve.

I asked Ethan about the features he would like to see in the chess app. He answered saying, "I would like to have an AI which properly simulates a realistic game that I would play against an actual opponent." This is reiterated as within my survey I asked potential users whether a chess AI would

appeal to them. The results are shown below:



Furthermore, when talking to Ethan about how I could direct my solution to a wide range of people. He said, “Being able to change the difficulty of the AI is a must, as to improve you need to be able to play against something of a similar skill level to you”. Therefore, in my solution, I aim to allow the user the ability to change the difficulty of the AI.

Another feature Ethan talked about was a chess puzzle feature within my solution. A chess puzzle is when you are presented with a chess position and the goal is to find the single best move or a series of moves which would put you in a winning position. He said, “By completing puzzles, I will be able to improve my tactical decision-making, ensuring I can take advantage when my opponent makes a mistake.” This is further backed up by the responses I received in my survey. Although when creating the survey, I didn’t think to include chess puzzles, users entered that they specifically would like to see “unique” and “free” puzzles. Therefore, for my solution, I aim to implement a chess puzzle feature.

Ethan also talked about how he would like to see the user interface of my solution to be “user-friendly, as it would appeal to the younger year students”. Therefore, within my solution, I aim to make it easy to use and navigate so that it can be enjoyed by a wide range of age groups.

I discussed with Ethan about whether he would like to see an account system within my solution. He said, “An account system would be necessary as it would allow me to track my progress and would allow me to review the game that I have played.” Therefore, within my solution, I aim to implement to ability for users to log in to their account allowing them to see details like their game history which would concern them and no one else.

Existing solutions

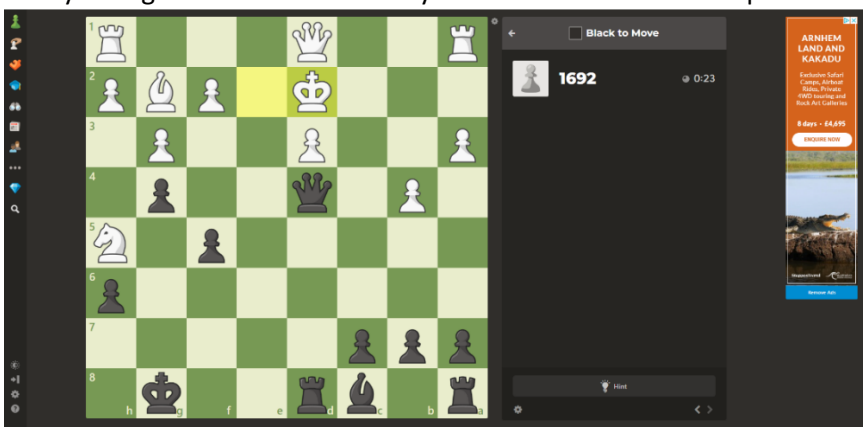
At present, there are solutions to the problem that I am tackling. One notable solution is Chess.com. Chess.com is used by many users as it allows users to connect and play against people. Chess.com is a well-rounded solution. It allows users to perform many different tasks enabling them to improve their game. The notable feature of Chess.com is the ability to play against people online, to complete puzzles and play against AI.

Within, my solution, I aim to implement features within Chess.com to allow students of the chess club to improve and refine their skills, whilst providing users with a good playing experience.

The picture below shows the interface when playing against an AI. Through this image and the experience, I had when I used the app, I learnt how they have made their user experience to be easy to use through having a very simplistic design. On the left column, they have a main menu which allows users to move between different sections of their app. To the right of the board, it displays the moves each player carries out and gives the option to users the ability to move back and forth within the game. Therefore, the chess application would allow users to analyse the position and how they got to that position mid-game. In addition, Chess.com allows the user to resign from the game so the user can end the game whenever they want to.

When playing, I noticed that it highlighted the square the moved piece was initially on and the square the piece ended up on. I would like to implement this feature as it allows users, if they get distracted, the ability to understand what happened and what moves their opponent has made.

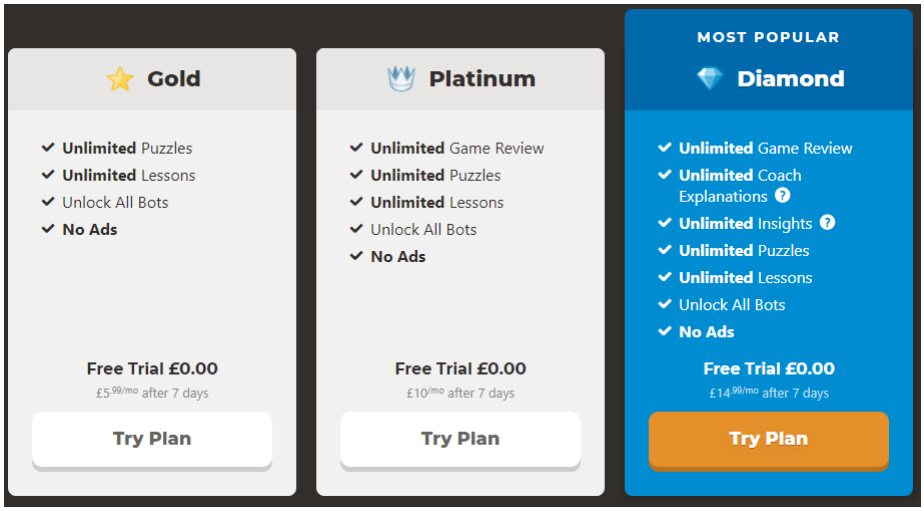
The picture below shows the interface when completing a chess puzzle. It uses the same design layout when playing against an AI. The chess position, rating and the colour to move are displayed. When solving, it also shows the time it takes to complete the puzzle. If the user is finding it difficult, the system gives the user the ability to have hints to solve the problem.



Problems with existing solutions

However, (as shown below) there is a subscription fee to access the premium features of Chess.com ranging from £5 - £15 per month. The main premium features include:

- The ability to complete chess puzzles whenever you want without a daily limit.
- The ability to play against all chess AIs.
- The ability to review your games.
- An ad free service.

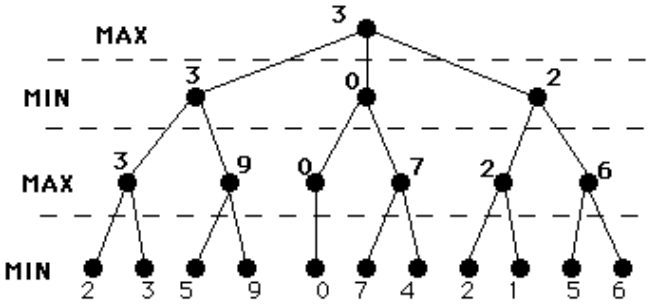


So, a free solution would be beneficial to students whilst still providing similar benefits.

How are Chess Engines implemented

After researching some chess engines, I discovered that a basic chess engine usually follows the same design: finding all the player's moves, iterating through the moves to a certain depth and finding the best moves by evaluating all the possible moves.

The most capable chess engine available to this date is Stockfish. Stockfish uses a combination of algorithms to examine and evaluate positions. The Minimax algorithm is a recursive algorithm used to find the best move. Minimax works by calling itself until it reaches a certain depth. Then at the maximum depth, it calculates a static evaluation of that end position. Then it returns up the tree where it alternates between the maximising and the minimising player. For the maximising player, the best move for the player would be the move with the highest evaluation of its branches. For the minimising player, the best move for the player would be the move with the lowest evaluation of its branches.



Chess engines, like Stockfish, utilise various optimisation technique to allow for the AI to search faster and search at a higher maximum depth. I have listed a few techniques below.

Optimisation technique	How does it work
Alpha-Beta Pruning	Tries to decrease the number of useless branches that are evaluated by the minimax algorithm.
Iterative Deepening	Carries out a search at a depth of 1 and then increments the search depth and does another search. This process is repeated until it exceeds a maximum time. Therefore, the program also has an option to fall back on. Each search should use the best move found by the previous search, thus maximising the number of nodes pruned.
Move Ordering	Sorts the moves so the best move is searched first. Thus, maximising the number of nodes pruned.
Transposition Tables	Uses a hash table that's stores the results of previously performed searches for different chess positions. Therefore, reducing the number of moves that must be made and unmade, thus significantly reducing processing time.

Chess engines need to be able to evaluate a chess position accurately to ensure it can accurately determine whether the position favours white or black. There are many factors which are used by chess engines to evaluate the position of the board. Below I have listed a couple which are utilised by chess engines.

Methods used to evaluate a position	How it works
Material imbalance	Calculating the total value of the pieces for both players
Positional advantage	Having pieces and on and controlling certain squares
Positioning of Pawns	Determining whether the player's pawn is in an advantageous position for example strong pawn structure or it being a passed pawn
Threats	Checks for whether the king is under threat
King Safety	Determines whether the king is in a safe position

Therefore, for my own chess AI I aim to use a minimax algorithm, some of the techniques used to evaluate the chess position and optimise the chess AI.

Summary of findings

After completing my research, I have found out that through the end-user, my survey, and an existing solution the following:

- Users would like to see a solution that provides user the ability to complete puzzles.
- Users would like to see a solution that provides user the ability to play against the AI.
- Users would like an AI which matches their skill level to refine their chess skills.
- Users would like an account system as it would allow them to track and save their progress through the puzzles and their game history.
- Users would like to see a simple user interface.

Modelling

Overview of the system – A move when playing against a Chess AI

Input	The user will move a piece from 1 square to another on the board
Process	The system will determine whether the user has made a valid move. Using the move, the program determines the best possible move for it to play, using a minimax algorithm. The system will convert the move made to chess notation. The system will determine whether the player/ AI is in check, checkmate or they are in a draw situation
Storage	Store the moves made by both the user and the AI in chess notation
Output	Output the new position of the piece on the board. Remove any pieces on the board that has been taken and move the icon the side of the board. Update the move log adding the new moves that has been made. Every second update the timer for the player when player in making a move.

Overview of the system – A move when completing a puzzle

Input	The user will move a piece from 1 square to another on the board
Process	Compare whether the move made by the user is the same as the predetermined best move in the database data for the puzzle
Storage	If the answer is correct, store the time taken and that the puzzle was completed by the user
Output	Display “Congratulation” message if the answer is correct and a “Try Again” message is the answer is wrong and reset the chess position if incorrect.

Rules of chess

How to set up a chess game

Chess is played on a board made up of 8x8 squares. The squares produce a chequer pattern where the white square being the rightmost square along the edge closest to each player.

The player's pieces are placed on the 2 ranks closest to the player. The closest rank consists of:

- 2 Rooks in the corner squares.
- Knights on the inside space next to the rooks.
- Bishops on the inside space next to the knights.
- The queen is placed on the remaining square which matches her colour.
- The king is placed on the remaining square.

The second rank from the player's perspective consists of 8 pawns, where each pawn is placed on a single square.

The image below shows a correctly set-up chess board.

How to move each chess piece

King

- The king can only move 1 square in any direction.
- The king cannot move into check. This is where the move could lead the king being captured by the opposition in the next more.

Queen

- The queen can move as many squares as possible in any direction.
- The queen cannot move through any pieces of the same colour.

Bishop

- Bishops can move as many squares as possible but can only move diagonally.
- Bishops cannot move through any pieces of the same colour.

Rook

- Rooks can move as many squares as possible but can only move vertically or horizontally.
- Rooks cannot move through any pieces of the same colour.

Knight

- Knights move in an L shape (2 squares in one direction, and 1 move perpendicular to the initial direction).
- Knights can move over other pieces.

Pawn

- Pawns may move two squares forward, if the pawn has not yet moved. Anywhere else, the pawn can only move in a straight line.
- Pawns may move diagonally when capturing an opponent's piece if an opponent's piece is situated on either the diagonal spaces to the left or right ahead of the piece.

Pawn Promotion

When a pawn reaches the other side of the board it can become any other chess piece excluding a king.

En Passant

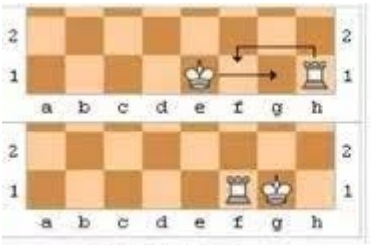
An "en passant" occurs when a pawn moves out two squares on its first move. If there is an opponent pawn next to the resulting pawn square, the opponent can capture the pawn. This capture can only occur immediately after the first pawn has moved.

Castling

The purpose of castling is to move the king into safety and to get the rook out of the corner so it could be utilised. When castling, the king moves 2 squares in the direction towards the rook and the rook moves to the square the king moved "through". To castle these conditions must be met:

- It must be the king's first move.
- It must be the rook's first move.
- The squares between the rook and the king must be empty.
- The king must not be in check or pass through check when castling.

There are two types of castling: Queen-Side and King-Side Castling. Below shows the difference between the 2 types.


King Side Castle


Queen Side Castle

Check

"Check" is when a piece moves so that the opponent's king is under-threat (where he could be captured). Therefore, the opponent's next move must be to move the king out "check" or move another piece to stop the attack.

Checkmate

"Checkmate" is when the king is in "check" and there are no possible moves to stop the king from being captured on the next turn. So, the player who got the other player in "checkmate" wins.

Draw

A draw is when there is no winner. This occurs when the position reaches stalemate, the players agree to a draw, there are not enough pieces on the board to force a checkmate and the same exact position is repeated three times. Stalemate is when it is a player's turn to move, and the player does not have another legal move to play and is not in check.

Chess notation

Chess notation is a method used to record the moves made in a chess game. The standard form of chess notation is called algebraic notation.

Below I have described how chess notation is structure:

Every piece, except for the pawns, have a character which refers to each piece.

Rook = R

Knight = N

Bishop = B

King = K

Queen = Q

If the move does not take an opposition piece, only the end square is added, e.g. "Nf3" for a knight moving to the square f3 or "e4" when a pawn moves to the square e4. Whereas, if the move does take an opposition piece, a "x" is placed between the piece and the end square position in the chess notation, e.g. "Nxf3". When a pawn captures a piece, due to the pawn not having a character which refers to the piece, it uses the file that the pawn started on followed by "x" and the end square, e.g. "cxe4".

Special Notation:

There are some moves in chess which have its own notation. Below I have listed all the moves and their notations.

Queen-Side Castling - "O-O-O"

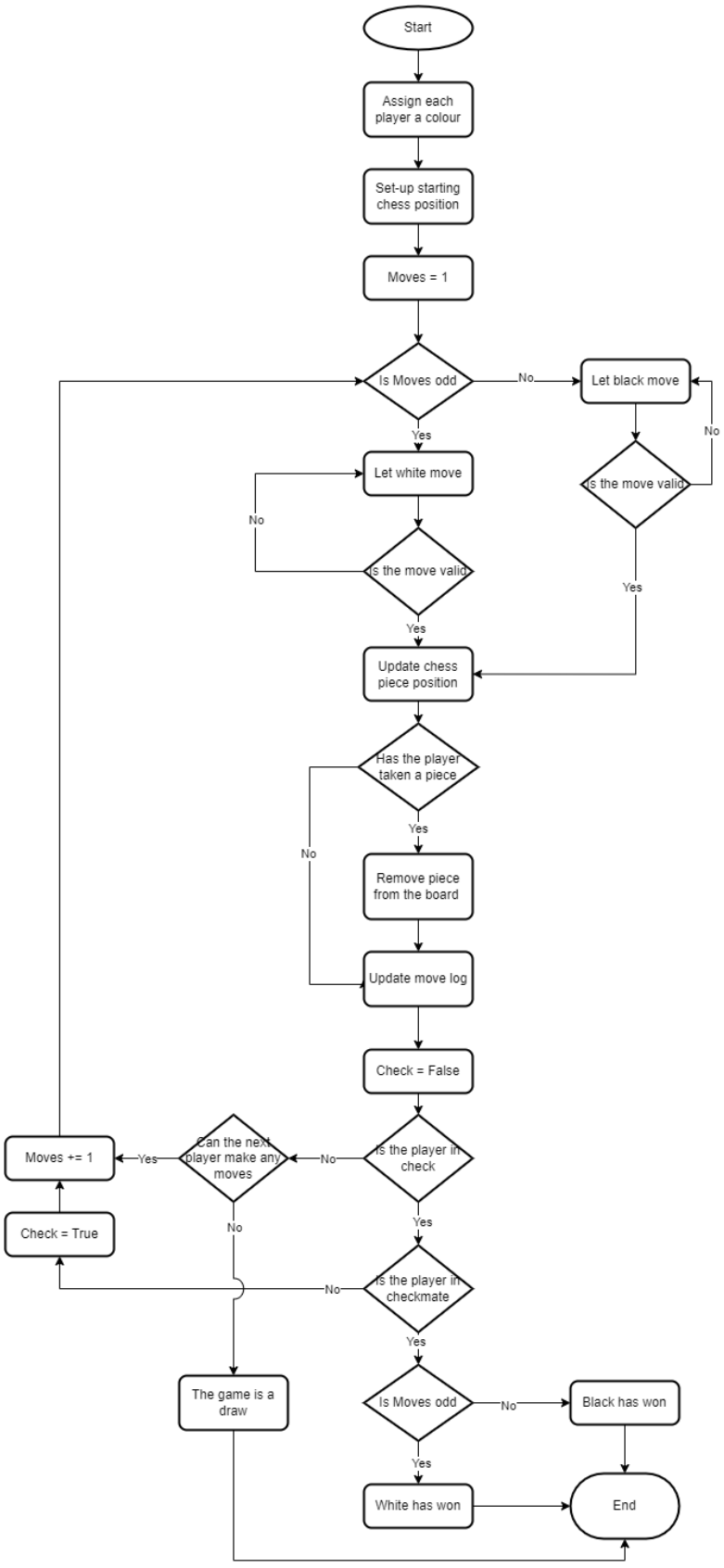
King-Side Castling - "O-O"

When the move causes the king to be in check - "+" is added to the end of the move.

When the move causes the king to be in checkmate - "++" is added to the end of the move.

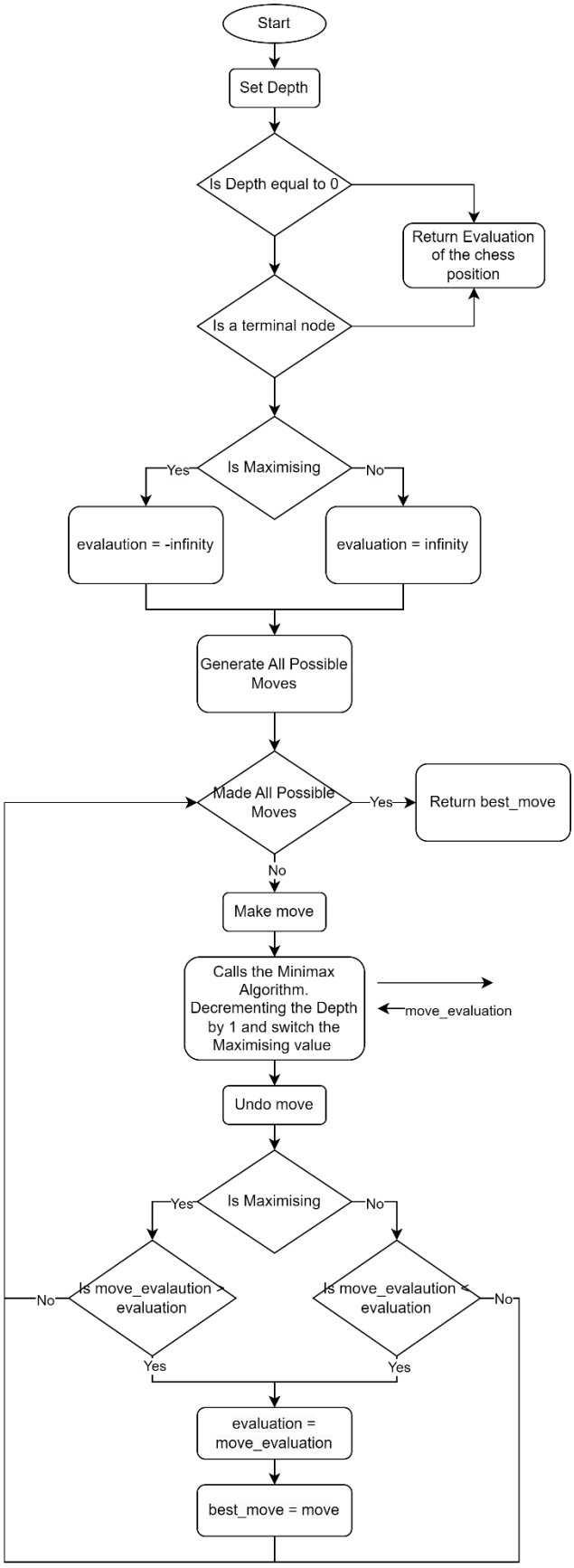
Flowchart of a chess game

Here I have shown what I believe to be the order of events and decisions of a typical chess game.

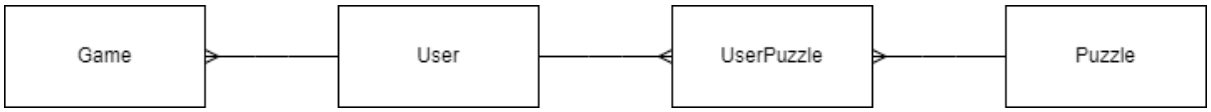


Flowchart of the Minimax Algorithm

Here I have shown what I believe how the chess engine will operate using the Minimax Algorithm.



Normalisation of my database



The User table will have attributes that stores details about the user like their username and password.

The Puzzle table will have attributes that stores details about the puzzle like rating, the chess position, and the correct move.

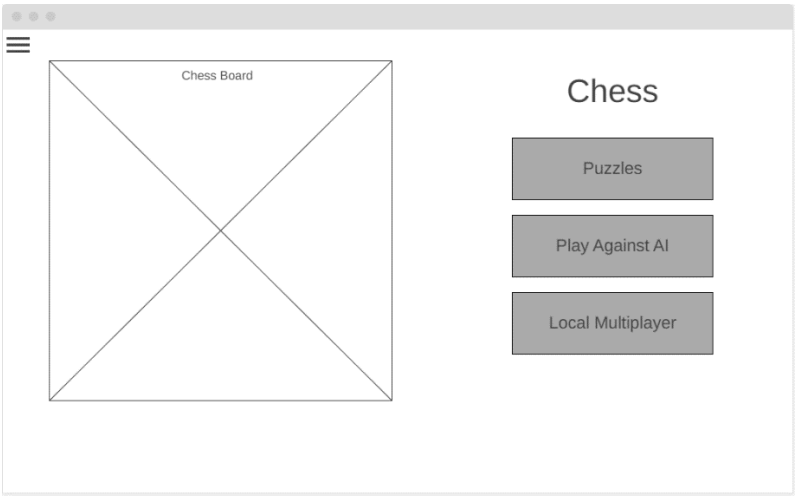
The UserPuzzle table will contain a composite primary key of the primary keys of the User and Puzzle table. This table will link the user and the puzzles they have solved.

The Game table will store the games the user has played against the AI so will have attributes like the move log, who won and difficulty of the AI.

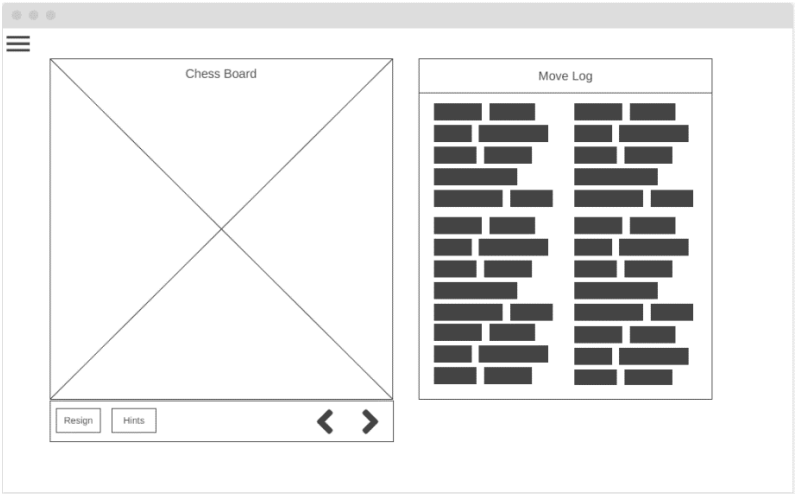
Mock-up user interface

Below, I have designed some preliminary designs for my user-interface of my system.

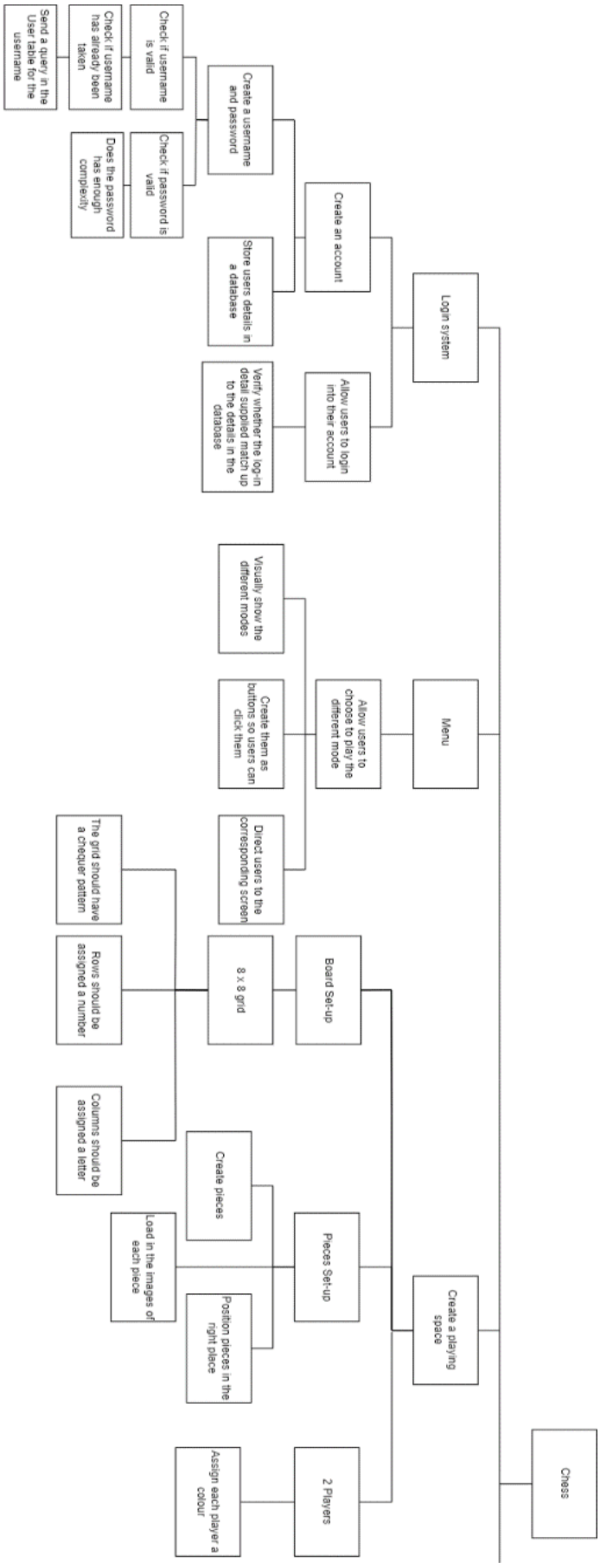
Design when in the main menu:

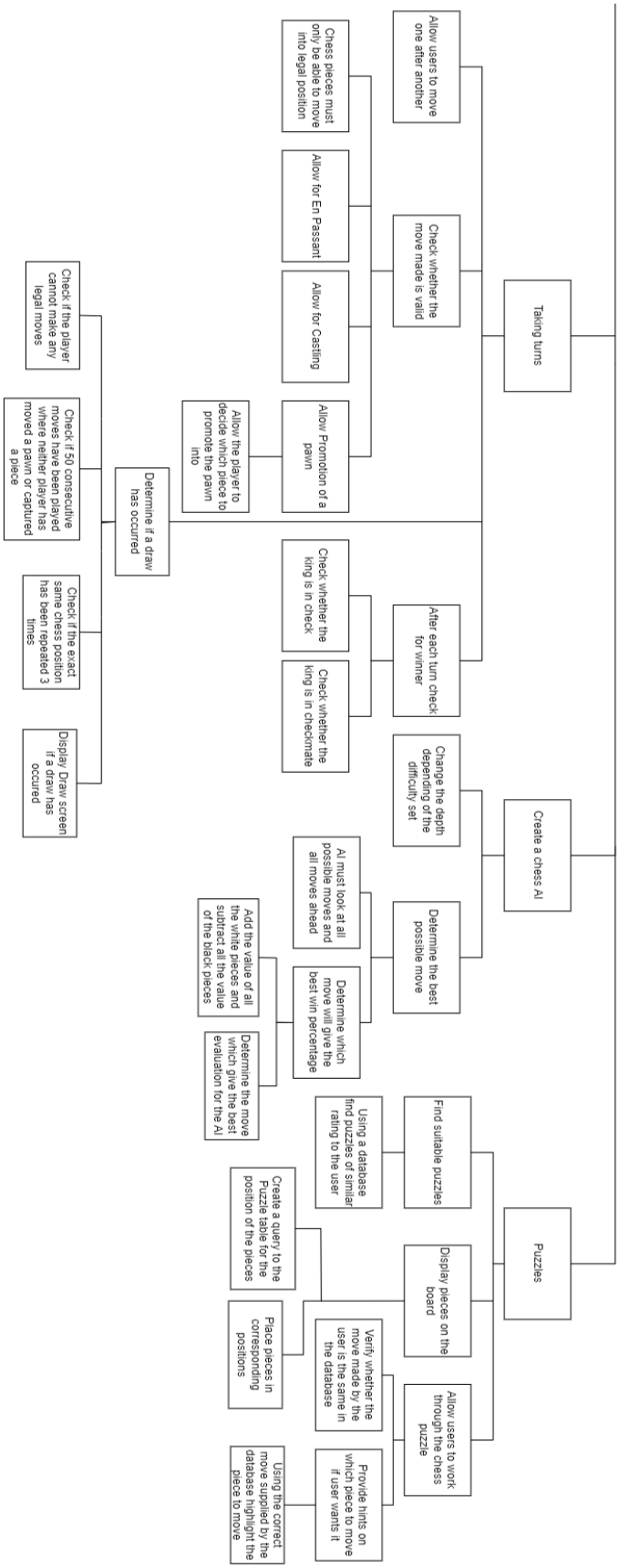


Design when playing a game of chess:

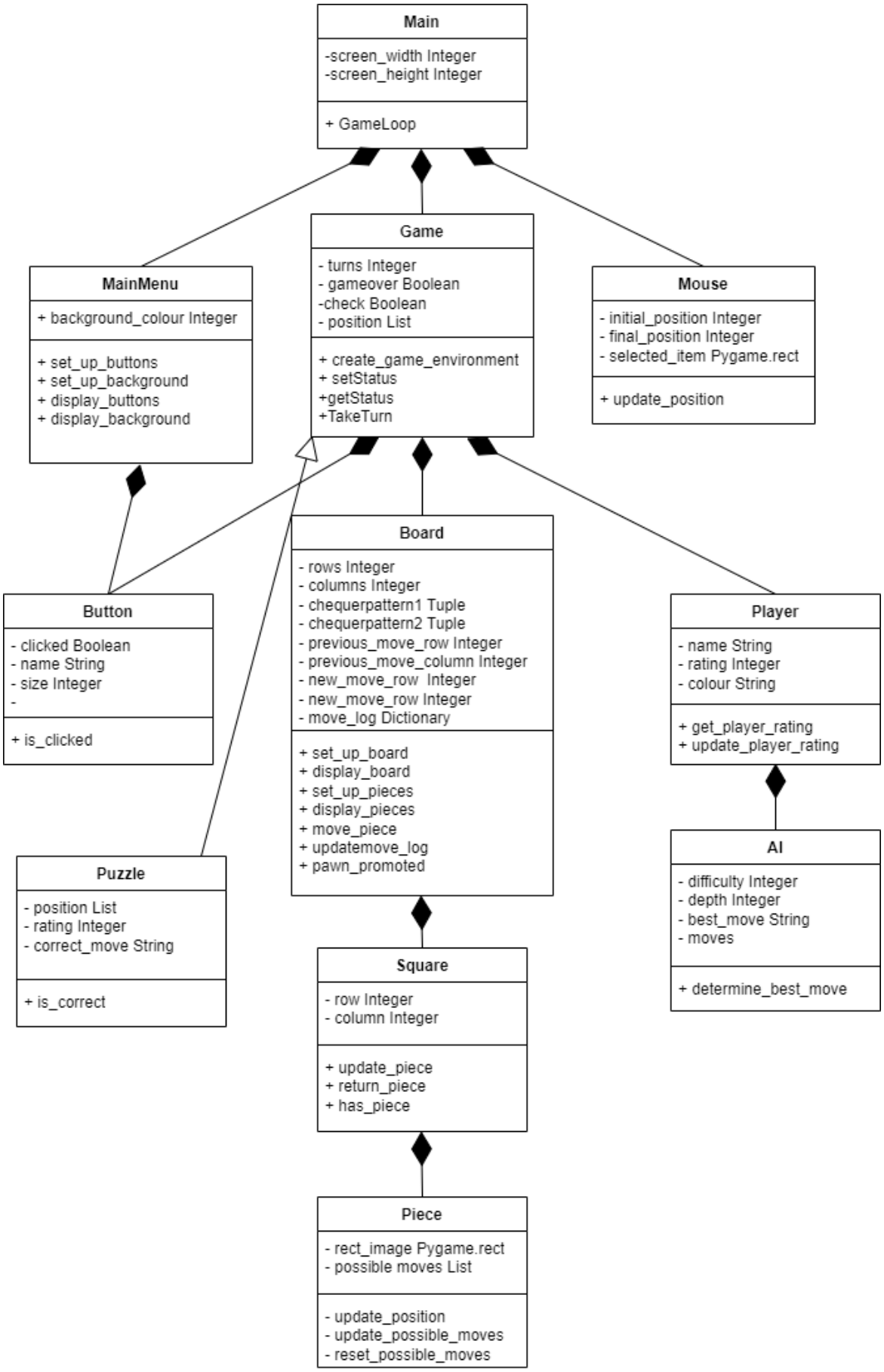


Preliminary Decomposition of my system




Taking turns
Allow users to move one after another
Check whether the move made is valid
Chess pieces must only be able to move into legal position
Allow for En Passant
Allow for Castling
Allow Promotion of a pawn
Allow the player to decide which piece to promote the pawn into
After each turn check for winner
Check whether the king is in check
Check whether the king is in checkmate
Determine if a draw has occurred
Check if the player cannot make any legal moves
Check if 50 consecutive moves have been played where neither player has moved a pawn or captured a piece
Check if the exact same chess position has been repeated 3 times
Display Draw screen if a draw has occured
Create a chess AI
Change the depth depending of the difficulty set
Determine the best possible move
AI must look at all possible moves and all moves ahead
Determine which move will give the best win percentage
Add the value of all the white pieces and subtract all the value of the black pieces
Determine the move which give the best evaluation for the AI
Puzzles
Find suitable puzzles
Using a database find puzzles of similar rating to the user
Display pieces on the board
Create a query to the Puzzle table for the position of the pieces
Place pieces in corresponding positions
Allow users to work through the chess puzzle
Verify whether the move made by the user is the same in the database
Provide hints on which piece to move if user wants it
Using the correct move supplied by the database highlight the piece to move

Preliminary UML of my system



Objectives

1) The system must allow users to register for an account.
   a) The system must confirm whether the username entered is unique.
   b) The system must determine whether inputted password is secure enough.
      i) The system must only allow passwords longer than 7 characters.
      ii) The system must only allow passwords containing a special character.
2) The system must allow users to log into their account.
   a) The system must verify that an account is associated with the username.
   b) The system must verify that the entered password is the same as the password in the database.
3) The system must allow users to navigate the system.
   a) The system's buttons must carry out its intended function.
      i) The "Puzzles" button must take users to the puzzle section of the system.
      ii) The "Play Against AI" button direct users to be able to play against the AI.
      iii) The "Play Against Human" button directs users to be able to play against another person.
      iv) The "Back" button directs users back to their previous screen.
4) The system must follow the rules of chess.
   a) The system must make white move first.
   b) The system must make users take turns when making moves.
   c) The system must only allow pieces to move in their set ways.
      i) The system must only allow the pawn to move 2 squares if it's the pawn's first move and the 2 squares ahead of the pawn are empty.
      ii) The system must only allow the pawn to move 1 square if the square ahead of the pawn is empty.
      iii) The system must only allow the pawn to move 1 square diagonally ahead if the square diagonally ahead contains an opposition piece.
      iv) The system must only allow the knight to move in an L shape if the end square is empty or contains an opposition piece.
      v) The system must only allow the bishop to move diagonally until it is blocked by one of its own pieces or is able to take an opposition piece.
      vi) The system must only allow the rook to move vertically or horizontally until it is blocked by one of its own pieces or is able to take an opposition piece.
      vii) The system must only allow the queen to move vertically, horizontally or diagonally until it is blocked by one of its own pieces or is able to take an opposition piece.
      viii) The system must only allow the king to move 1 square in all directions if the square is not being targeted by an opposition piece and if the square is empty or contains an opposition piece.
   d) The system must allow users to perform an En Passant if these conditions are met:
      i) The system must check that the opposition player moves the pawn by two squares.
      ii) The system must check that the previous move was made by the pawn.
      iii) The system must check that the end square of the opposition player's pawn is directly next to the player's pawn.
   e) The system must allow users to perform a King and Queen side castle if these conditions are met:
      i) The system must check that the squares between the king and the rook are empty.
      ii) The system must check that the king and the rook have not moved previously.

   iii) The system must check that the squares the king would move through would not put the king in check.
   iv) The system must check that the king is not in check initially.
   f) The system must allow for a pawn promotion if the pawn has made it to its 8th rank the board.
   g) The system must not allow the player to make a move that will cause the king to be in check.
   h) The system must determine when a king is in check.
   i) The system must ensure that the next move played by the player removes the check.
   j) The system must determine when a king is in checkmate.
   k) The system must determine when a player is in stalemate.
   l) The system must determine when there is a draw by repetition.
5) The system must allow users to be able to play chess.
   a) The system must position the pieces in the correct squares.
   b) The system must visually show the board to the user.
   c) The system must track the moves made in the game in chess notation.
   d) The system must allow users to be able to move back and forth within the game.
   e) The system must highlight the previous move and the start and end square of the piece.
   f) The system must update the chess piece when a move is made.
6) The system must allow users to play against an AI.
   a) The system must be able to change the difficulty of the AI and the colour to play as.
   b) The system must make the AI play legit moves.
   c) The system must ensure the AI provide users with acceptable response times.
7) The system must provide users the ability to review previously played games.
   a) The system must allow users to save games they have played.
   b) The system must allow users to filter the games that are saved.
   c) The system must allow users to retrieve the game.
   d) The system must allow users to progress through the game.
   e) The system must allow users to be able to try different moves than the moves they played in the game.
8) The system must provide users the ability to complete chess puzzles.
   a) The system must allow users to select the range of puzzles that the user want to complete.
   b) The system must visually show the chess position of the pieces.
   c) The system must show the rating of the puzzle.
   d) The system must show to the user the puzzle is complete if the user enters the correct move.

Possible solutions

Programming Language

There are many high-level programming languages that I could use that would achieve my goal. However, I have the most experience using Python compared to other programming languages, and because Python is also easy to write and understand, I have decided to use Python.

UI Library

For my solution, I have investigated library to use when programming my solution. There are several libraries that would provide me the functionality that I need. The two main libraries that I could utilise are PyGame or Arcade. When following tutorials, both libraries are easy to use but I found that Arcade is easier to program compared to PyGame, when achieving the same solution. However, after reading a review about Arcade, I discovered that the main disadvantage of using Arcade is that it is a newer library therefore a greater possibility of incurring bugs when programming. This is compared to PyGame which has been around for over 20 years, so the library has been tested over a longer period and so fewer chances of problems occurring when programming my solution.

Server-Side

I intend to use PythonAnywhere to hold my database as it will allow me to separate the programming of the server-side of my solution from my client-side. In addition, it will reduce the number of instructions carried out on the client device making my application faster to run. PythonAnywhere provides a free service allowing users to use a limited version of an industry standard Python environment. This free version suffices for my intended use, so PythonAnywhere is an appealing choice.

Design

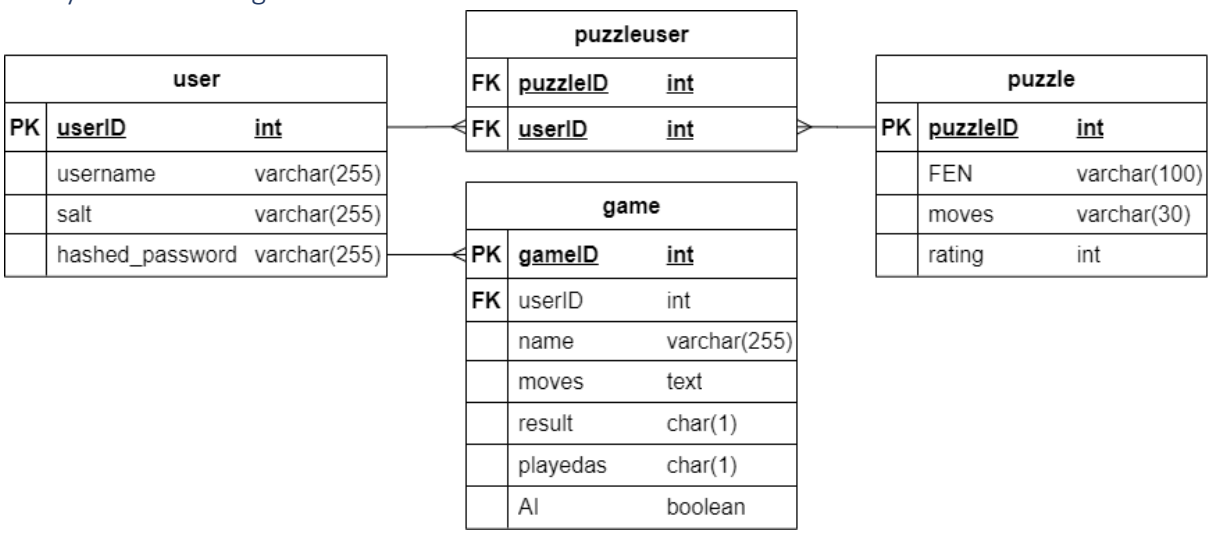
Overview

My application contains four main sections to help users improve their chess ability. These sections include playing against a human locally, playing against an AI, completing puzzles and reviewing previously played games.

Database Design

I required to use a database to store information about the users. So, the database is users to log in to the application and track progress users have made through the puzzles and store game the user has played.

Entity Attribute Diagram



The “user” table:

When a user creates their account, the user sets a username and password. If the username is not already being used by another user, then the username is stored in the database. I did not store the password as a plaintext in the database, as this would compromise the security. Instead, I used a large, randomised value, called a salt, which is generated on the user’s device. This salt is then hashed with the plaintext of the password. The hashed password and the salt are then stored with the user’s details.

The “game” table:

When a user saves a game, it creates a record in the “game” table. The table has a one-to-many relationship with the “user” table, as each user can save multiple games however each game can only correspond to one account. The user creates a name for the game and details about the result, who the user played as, if the game was against an AI and the moves played, as coordinates on the board, are stored.

The “puzzle” table:

Using a database of chess puzzles, I extracted the FEN, moves and rating for each puzzle and uploaded it to the “puzzle” table in my server.

The “puzzleuser” table:

I used a linking table which used the two foreign keys of both the “user” and the “puzzle” tables to create a composite primary key, thus creating a link between the user and the puzzle.

The database contains any partial or non-key dependencies. Therefore, the table is in third normal form.

Server

I have used Flask to create a web server to handle user requests for data held in the server. In addition, I used Flask App Routing to create a web framework. It works by mapping a URL to a specific function. Using Flask App Routing, I have also specified the HTTP methods each function should respond to. I used the library MySQLdb to communicate with the database. In addition, I have used JSON to format the data, as data is passed from the client to the server.

The HTTP methods I have used are POST and GET and they are typically implemented using the following techniques:

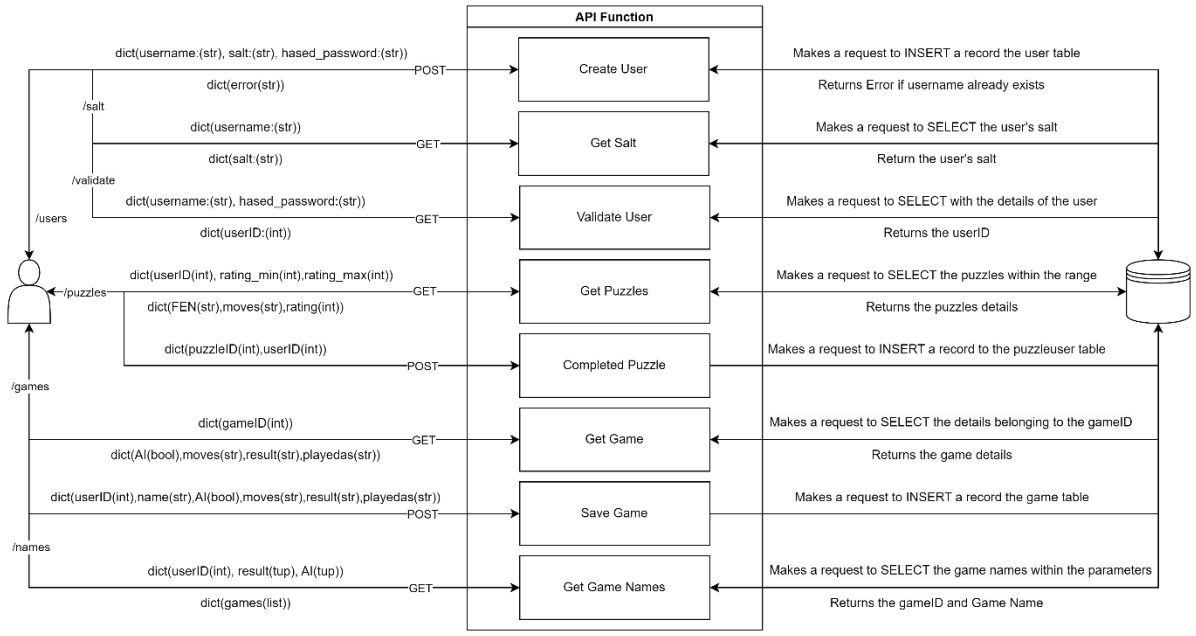
POST:

The function requests for the JSON data. Then the function the connects to the database. A cursor is then used to execute the SQL commands to insert the values into the table in the database. Then the changes are committed to the database and then the database connection is closed. If there was an error during the execution of the SQL command, it returns a JSON response with both an error message and status code.

GET:

The function request for the arguments. Then the function the connects to the database. A cursor is then used to execute the SQL commands to select the values from the table in the database. The database connection is closed. The function then returns the fetched data to the client as a JSON response.

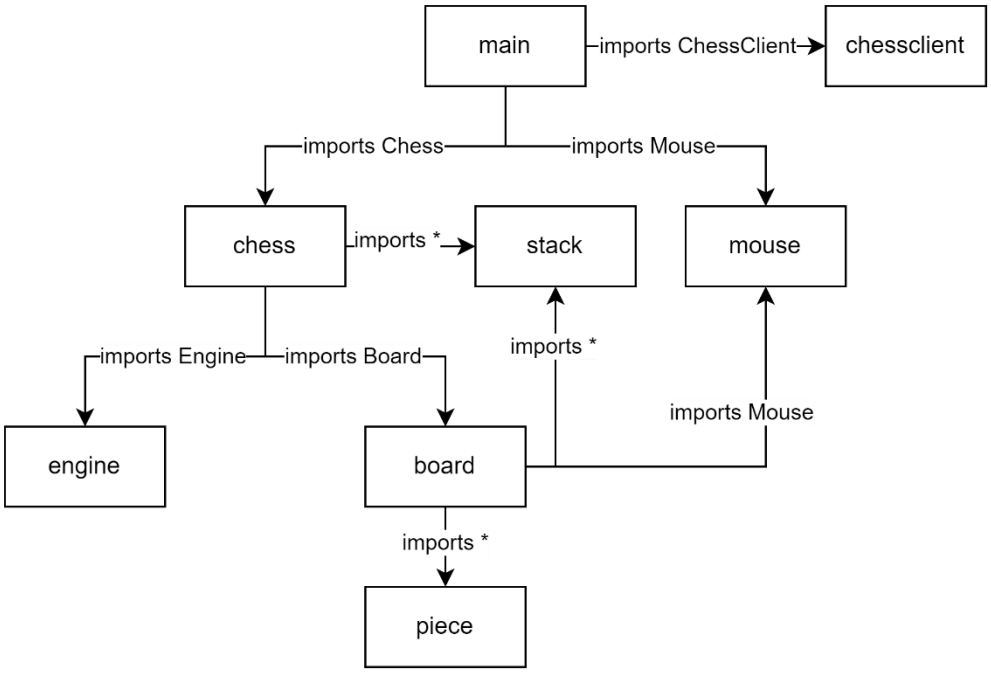
The diagram below describes how the client communicates with the server:



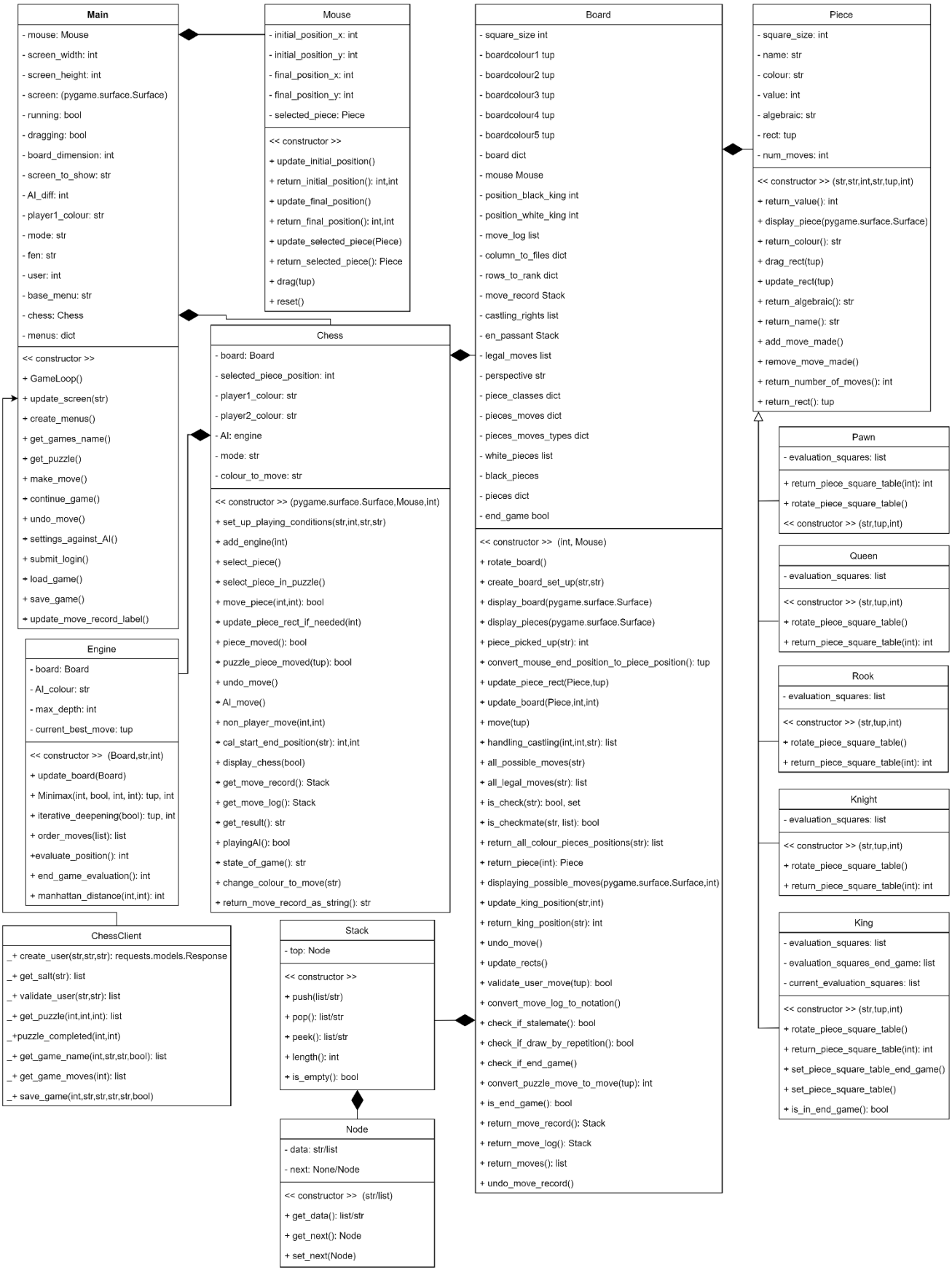
OOP and Programming

File Structure Diagram

In the below diagram, it describes how the files of the application are structured. Therefore, it shows the classes each file imports from the other files where the arrow is pointing to the file which contains the class being imported.



Final UML


Main
- mouse: Mouse
- screen_width: int
- screen_height: int
- screen: (pygame.surface.Surface)
- running: bool
- dragging: bool
- board_dimension: int
- screen_to_show: str
- AI_diff: int
- player1_colour: str
- mode: str
- fen: str
- user: int
- base_menu: str
- chess: Chess
- menus: dict
<< constructor >>
+ GameLoop()
+ update_screen(str)
+ create_menus()
+ get_games_name()
+ get_puzzle()
+ make_move()
+ continue_game()
+ undo_move()
+ settings_against_AI()
+ submit_login()
+ load_game()
+ save_game()
+ update_move_record_label()
Mouse
- initial_position_x: int
- initial_position_y: int
- final_position_x: int
- final_position_y: int
- selected_piece: Piece
<< constructor >>
+ update_initial_position()
+ return_initial_position(): int,int
+ update_final_position()
+ return_final_position(): int,int
+ update_selected_piece(Piece)
+ return_selected_piece(): Piece
+ drag(tup)
+ reset()
Chess
- board: Board
- selected_piece_position: int
- player1_colour: str
- player2_colour: str
- AI: engine
- mode: str
- colour_to_move: str
<< constructor >> (pygame.surface.Surface,Mouse,int)
+ set_up_playing_conditions(str,int,str,str)
+ add_engine(int)
+ select_piece()
+ select_piece_in_puzzle()
+ move_piece(int,int): bool
+ update_piece_rect_if_needed(int)
+ piece_moved(): bool
+ puzzle_piece_moved(tup): bool
+ undo_move()
+ AI_move()
+ non_player_move(int,int)
+ cal_start_end_position(str): int,int
+ display_chess(bool)
+ get_move_record(): Stack
+ get_move_log(): Stack
+ get_result(): str
+ playingAI(): bool
+ state_of_game(): str
+ change_colour_to_move(str)
+ return_move_record_as_string(): str
Engine
- board: Board
- AI_colour: str
- max_depth: int
- current_best_move: tup
<< constructor >> (Board,str,int)
+ update_board(Board)
+ Minimax(int, bool, int, int): tup, int
+ iterative_deepening(bool): tup, int
+ order_moves(list): list
+evaluate_position(): int
+ end_game_evaluation(): int
+ manhattan_distance(int,int): int
ChessClient
_+ create_user(str,str,str): requests.models.Response
_+ get_salt(str): list
_+ validate_user(str,str): list
_+ get_puzzle(int,int,int): list
_+puzzle_completed(int,int)
_+ get_game_name(int,str,str,bool): list
_+ get_game_moves(int): list
_+ save_game(int,str,str,str,str,bool)
Stack
- top: Node
<< constructor >>
+ push(list/str)
+ pop(): list/str
+ peek(): list/str
+ length(): int
+ is_empty(): bool
Node
- data: str/list
- next: None/Node
<< constructor >> (str/list)
+ get_data(): list/str
+ get_next(): Node
+ set_next(Node)
Board
- square_size int
- boardcolour1 tup
- boardcolour2 tup
- boardcolour3 tup
- boardcolour4 tup
- boardcolour5 tup
- board dict
- mouse Mouse
- position_black_king int
- position_white_king int
- move_log list
- column_to_files dict
- rows_to_rank dict
- move_record Stack
- castling_rights list
- en_passant Stack
- legal_moves list
- perspective str
- piece_classes dict
- pieces_moves dict
- pieces_moves_types dict
- white_pieces list
- black_pieces
- pieces dict
- end_game bool
<< constructor >> (int, Mouse)
+ rotate_board()
+ create_board_set_up(str,str)
+ display_board(pygame.surface.Surface)
+ display_pieces(pygame.surface.Surface)
+ piece_picked_up(str): int
+ convert_mouse_end_position_to_piece_position(): tup
+ update_piece_rect(Piece,tup)
+ update_board(Piece,int,int)
+ move(tup)
+ handling_castling(int,int,str): list
+ all_possible_moves(str)
+ all_legal_moves(str): list
+ is_check(str): bool, set
+ is_checkmate(str, list): bool
+ return_all_colour_pieces_positions(str): list
+ return_piece(int): Piece
+ displaying_possible_moves(pygame.surface.Surface,int)
+ update_king_position(str,int)
+ return_king_position(str): int
+ undo_move()
+ update_rects()
+ validate_user_move(tup): bool
+ convert_move_log_to_notation()
+ check_if_stalemate(): bool
+ check_if_draw_by_repetition(): bool
+ check_if_end_game()
+ convert_puzzle_move_to_move(tup): int
+ is_end_game(): bool
+ return_move_record(): Stack
+ return_move_log(): Stack
+ return_moves(): list
+ undo_move_record()
Piece
- square_size: int
- name: str
- colour: str
- value: int
- algebraic: str
- rect: tup
- num_moves: int
<< constructor >> (str,str,int,str,tup,int)
+ return_value(): int
+ display_piece(pygame.surface.Surface)
+ return_colour(): str
+ drag_rect(tup)
+ update_rect(tup)
+ return_algebraic(): str
+ return_name(): str
+ add_move_made()
+ remove_move_made()
+ return_number_of_moves(): int
+ return_rect(): tup
Pawn
- evaluation_squares: list
+ return_piece_square_table(int): int
+ rotate_piece_square_table()
<< constructor >> (str,tup,int)
Queen
- evaluation_squares: list
<< constructor >> (str,tup,int)
+ rotate_piece_square_table()
+ return_piece_square_table(int): int
Rook
- evaluation_squares: list
<< constructor >> (str,tup,int)
+ rotate_piece_square_table()
+ return_piece_square_table(int): int
Knight
- evaluation_squares: list
<< constructor >> (str,tup,int)
+ rotate_piece_square_table()
+ return_piece_square_table(int): int
King
- evaluation_squares: list
- evaluation_squares_end_game: list
- current_evaluation_squares: list
<< constructor >> (str,tup,int)
+ rotate_piece_square_table()
+ return_piece_square_table(int): int
+ set_piece_square_table_end_game()
+ set_piece_square_table()
+ is_in_end_game(): bool

Algorithms

Below I have created pseudocode for complex algorithms in my program.

Generating All Legal Moves

In my application, to generate all the legal moves the player can make, I firstly find all the pseudo-legal moves. A pseudo-legal move is all the possible moves the player's pieces can make. The algorithm I have used to generate the pseudo-legal moves is below:

```
FUNCTION generate_all_possible_moves(colour)
    possible_moves = []
    IF colour_at_the_bottom_of_board == colour THEN
        upper_bound, lower_bound, multiple = 7, 0, 1
    ELSE THEN
        upper_bound, lower_bound, multiple = 0, 7, -1
    ENDIF

    FOR start_position IN pieces_position[colour]:
        piece = board[piece_position]
        piece_name = piece.return_name()
        move = pieces_moves[piece_name]
        FOR move IN moves:
            end_position = start_position + multiple * move
            IF 0 <= end_position <= 63 THEN
                target_piece = board[end_position]
                IF piece_name == "Pawn" THEN
                    row = start_position // 8
                    column = start_position % 8
                    IF move == N AND target_piece == None THEN
                        possible_moves.append((start_position,end_position))

                        IF (row == 6 AND colour == colour_at_the_bottom_of_board) OR (row == 1 AND
colour != colour_at_the_bottom_of_board) THEN
                            possible_moves.append((start_position,end_position))
                        ENDIF
                    ELIF (target_piece != None AND target_piece.return_colour() != colour) OR
is_en_passant_possible(end_position) THEN
                        IF move == N+E AND ((column < 7 AND multiple == 1) OR (0 < column AND multiple
== -1)) THEN
                            possible_moves.append((start_position,end_position))
                        ENDIF
                        IF move == N+W AND ((0 < column AND multiple == 1) OR (column < 7 AND multiple
== -1)) THEN
                            possible_moves.append((start_position,end_position))
                        ENDIF
                    ENDIF
                    continue
                ENDIF
                check = start_position
                WHILE True:
                    IF (check //8 == lower_bound AND N IN move) OR (check //8 == upper_bound AND S IN
move) OR (check %8 == lower_bound AND W IN move) OR (check %8 == upper_bound AND E IN move) THEN
                        BREAK
                    ENDIF
                    IF NOT(0 <= end_position <= 63) THEN
                        BREAK
                    ENDIF

                    target_piece = self.__board[end_position]
                    IF algebraic == "N" THEN
                        IF (check //8 == lower_bound+multiple AND N+N IN move) OR (check %8 ==
upper_bound-multiple AND E+E IN move) OR (check %8 == lower_bound +multiple AND W+W IN move) OR (check
//8 == upper_bound -multiple AND S+S IN move) THEN
```

```
                        BREAK
                    ELIF target_piece == None OR target_piece.return_colour() != colour THEN
                        possible_moves.append((start_position,end_position))
                    ENDIF
                ENDIF

                IF target_piece == None THEN
                    possible_moves.append((start_position,end_position))
                    IF piece_name == "King" THEN
                        BREAK
                    ENDIF
                    check = end_position
                    end_position += multiple*move

                ELSE THEN
                    IF target_piece.return_colour() != colour THEN
                        possible_moves.append((start_position,end_position))
                    ENDIF
                    BREAK
                ENDIF
            ENDWHILE
        ENDIF
    ENDFOR
ENDFOR
RETURN possible_moves
```

However, a pseudo-legal move is not always a legal move as the move may leave the king in check. So, I must check whether the move leads to the king being in check. If the king is in check, the move is illegal and vice versa. The algorithm I have used to generate the legal moves are below:

```
FUNCTION generate_all_legal_moves(colour)
    legal_moves = []
    all_moves = all_possible_moves(colour)
    op_colour = "w" IF colour == "b" ELSE "b"
    FOR move IN all_moves:
        make_move(move)
        start_position,end_position = move[0],move[1]
        king_position = return_king_position(colour)
        all_op_moves_end_position = move[1] for move IN all_possible_moves(op_colour)

        IF king_position NOT IN all_op_moves_end_position THEN
            legal_moves.append(move)
        ENDIF
       undo_move()
    ENDFOR
    king_position = return_king_position(colour)
    IF queen_side_castling_possible(colour) THEN
        legal_moves.append(start_position, return_queen_side_castle_end_position(colour))
    IF king_side_castling_possible(colour) THEN
        legal_moves.append(start_position, return_king_side_castle_end_position(colour))
    RETURN legal_moves
```

The Chess AI

Minimax with alpha-beta pruning:

The function minimises the maximum the possible loss which can result from a choice a player makes. Alpha-beta pruning is an optimisation technique that looks to decrease the number of nodes, in this case chess positions, that are evaluated by the minimax algorithm.

```
FUNCTION Minimax(depth,maximising_player,alpha,beta)
    IF depth == 0 OR game_is_end() THEN
        RETURN evaluate_position()
    ENDIF
    colour = "w" IF maximising_player ELSE "b"
    moves = get_all_legal_moves_in_position(colour)
    IF maximising_player THEN
        max_evaluation = -infinity
        FOR move IN moves:
            make_move(move)
            evaluation = Minimax(depth-1,False,alpha,beta)
            undo_move()
            IF evaluation > max_evaluation THEN
                max_evaluation = evaluation
                best_move = move
            ENDIF
            alpha = max(alpha,max_evaluation)
            IF beta <= alpha THEN
                BREAK
        ENDFOR
        RETURN best_move
    ELSE:
        min_evaluation = infinity
        FOR move IN moves:
            make_move(move)
            evaluation = Minimax(depth-1,True,alpha,beta)
            undo_move()
            IF evaluation > max_evaluation THEN
                max_evaluation = evaluation
                best_move = move
            ENDIF
            beta = min(beta,min_evaluation)
            IF beta <= alpha THEN
                BREAK
        ENDFOR
        RETURN best_move
    ENDIF
```

In addition to alpha-beta pruning, I used Move Ordering and Iterative Deepening to increase the efficiency of the chess AI.

Iterative Deepening

Iterative Deepening works by iteratively calling the Minimax algorithm where it increments the depth that it is searched at until the time elapsed is greater or equal to the time limit. This is an optimisation technique as regardless of the time there will always have a best move to play. Also, in combination with Move Ordering it can reduce the number of nodes that are searched as typically the best move in the previous search is usually the best move at the next depth and so allowing for more branches to be pruned per search.

```
FUNCTION Iterative_Deepeing(maximising_player)
    current_best_move = None
    start_time = time()
```

```
    depth = 0
    WHILE time() - start_time < time_limit or depth == max_depth:
        current_best_move = Minimax(depth, maximising_player, -infinity, infinity)
        depth += 1
    ENDWHILE
    RETURN current_best_move
```

Evaluating Chess Positions

The evaluation function used by the chess engine takes the values of the piece and the value of the square for the piece and adds and subtracts it from the evaluation depending on the colour of the piece. The piece-square tables are a 1-dimensional list which gives an integer value for each position. This value is set by the desirability of the square for each piece.

In addition, an end game evaluation is used as the evaluation for the end game is slightly different as in the end game you want the king to occupy the centre squares and the distance between kings to be smaller whereas in the opening and the middle game you want the king to be well protected.

```
FUNCTION Evaluate_Position()
    IF white_is_in_checkmate() THEN
        RETURN -infinity
    IF black_is_in_checkmate() THEN
        RETURN infinity
    IF draw() THEN
        RETURN 0

    evaluation = 0
    positions = return_all_white_pieces_positions()

    FOR i IN positions:
        piece = return_piece_in_board(i)
        evaluation += return_value(piece)
        evaluation += return_square_value_for_piece(piece,i)
    positions = return_all_black_pieces_positions()
    FOR i IN positions:
        piece = return_piece_in_board(i)
        evaluation -= return_value(piece)
        evaluation -= return_square_value_for_piece(piece,63-i)
    IF is_end_game() THEN
        evaluation += end_game_evaluation()
    RETURN evaluation
```

SQL

Below I have outlined any complex SQL queries that I used:

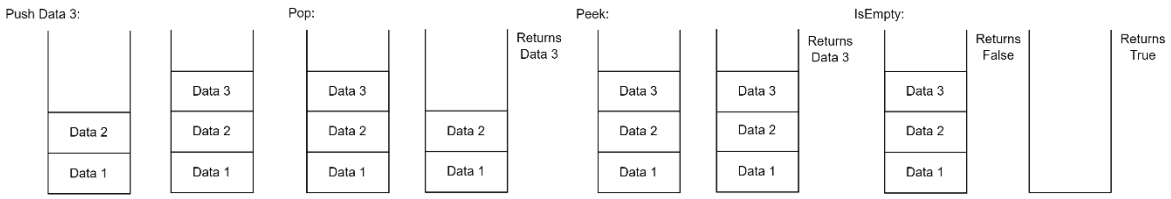
Selecting a Puzzle:

"SELECT * FROM puzzle WHERE rating = (SELECT MIN(rating) FROM puzzle WHERE rating < %s AND rating > %s AND puzzleID NOT IN (SELECT puzzleID FROM puzzleuser WHERE userID = %s)) AND puzzleID NOT IN (SELECT puzzleID FROM puzzleuser WHERE userID = %s) limit 1"

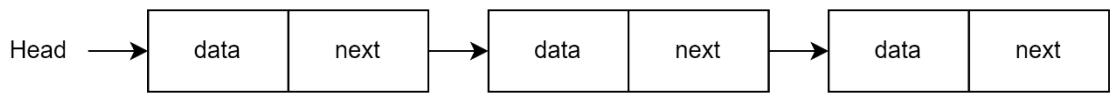
I have selected all the fields of the record in the puzzle table where the rating is between limits set by the user. In addition, using the aggregate SQL statement MIN, I select the puzzle with the minimum rating as you want the puzzles to get increasingly more difficult. In addition, when selecting a puzzle, you want the puzzle to be a puzzle that the user has not completed before. Therefore, I conducted a SQL query to select all the puzzles from the puzzleuser table where it has a relationship with the specific user. So, by using NOT IN it ensures that the selected puzzle is a puzzle which the user has not yet completed. I also had to repeat the same sub-query outside of the selection of the minimum rating to prevent a puzzle from being repeatedly picked as puzzles may have the same rating. I used a LIMIT clause to restrict the number of records which are selected, as a user completes one puzzle at a time.

Data Structure

In my application, I have utilised a stack, which is a First In Last Out data structure, to store the moves and en passant squares throughout the game. This works, as when a user or the AI makes a move, the details about the moves are pushed onto the top of the stack. These details include the start and end position, whether the move was a king/queen side castle, en passant or a capture move. So, if the player decides that they want to undo a move, the move at the top of the stack can be popped from the stack and the board can return to the original position. In addition, throughout my code, I have used other stack functions like Peek which returns the item at the top of the stack but not removing it and IsEmpty to check if the stack is empty.

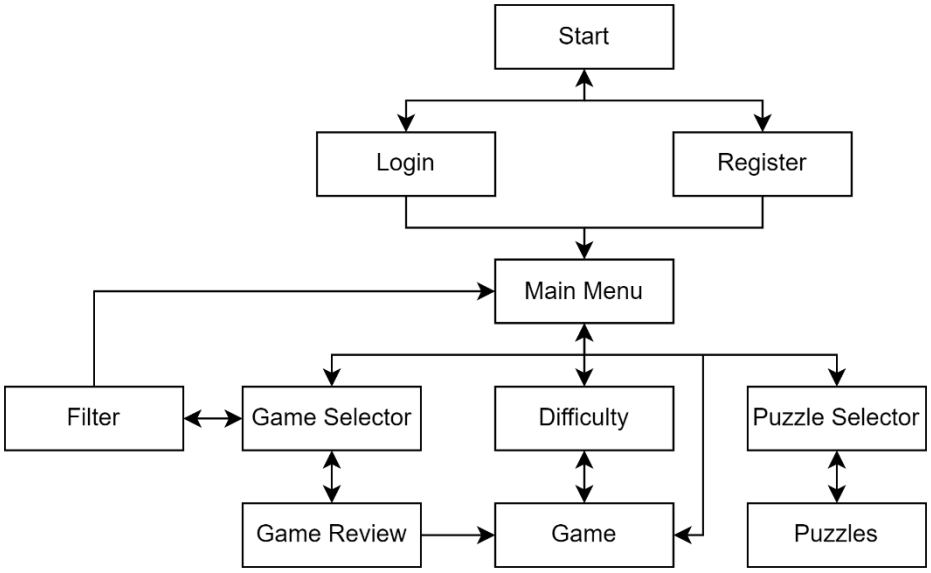


I have implemented the stack using a linked list. I did this as a linked list has a dynamic size and because the number of moves in the game is variable it allows the maximum stack size to dynamically change. The way a linked list works is that each node in the stack has two attributes which stores the data of the node and the reference to the next node in the list. Therefore, elements in the list are linked using pointers.



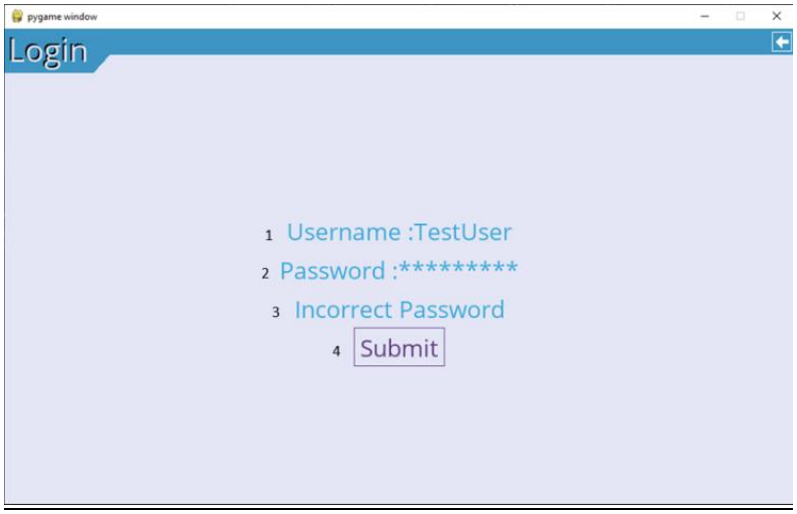
User Interface

Below describe how users navigate the menus of the application where the arrows are pointing to the menu it can access:



I used a Pygame library called Pygame-menu to create the menus for my application. I decided on using this as creating my own widget would be inefficient. Therefore, it allowed me to focus on the functionality of my application.

Log-in



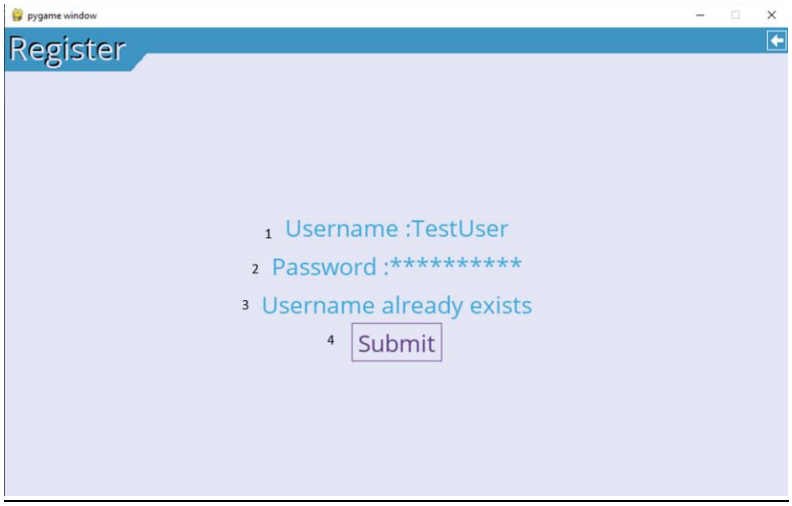
1 - A text input box that allows the user to input the username for their account.

2 - A text input box that allows the user to input the password for their account.

3 – A label that is updated when the inputted data are invalid.

4 - A button that allows the user to log into their account if the supplied details are correct.

Register



1 - A text input box that allows the user to input a username.

2 - A text input box that allows the user to input a password.

3 – A label that is updated when the inputted data are invalid.

4 - A button allows the user to create an account if the supplied details are valid.

Main Menu

1 – A button that takes the user to the “Difficulty” Menu

2 – A button that takes the user to the “Game” Menu

3 - A button that takes the user to the “Selector” Menu

4 - A button that takes the user to the “Game Selector” Menu

5 - A button that exits the application

Playing against a Human or AI

When playing, I split up the screen, so the board fills up the left-hand side of the screen and the menu fills what is remaining. When it's the player's turn, the user picks up a piece by clicking the piece, to move the piece the user drags and drops it in the desired end position.

1 - The Move Log is updated to display the move made in chess notation.

2 - If a move has caused a check, checkmate or a draw the label is updated to notify the user.

3 - A button that allows the user to undo the previous moves that have been made.

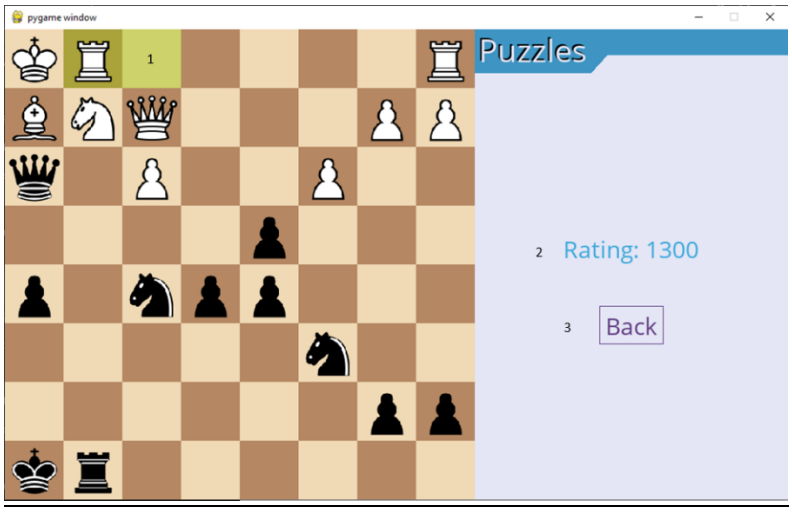
4 – A button that allows the user to exit the game

5 - A text input box that allows the user to input a name they would like to call the chess game.

6 - A button that stores the details of the game onto the database.

7 - After each move, the start and end location of the piece that was moved is highlighted.

<u>Completing Puzzles</u>



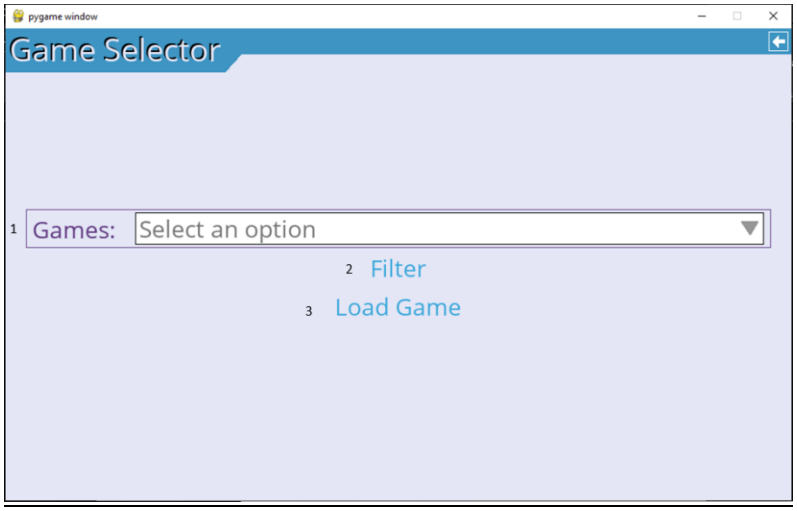
The application displays the puzzle position where the user pieces start from the bottom of the board.

1 - The move made by the opposition is highlighted. Therefore, allowing users to interpret the game and make a move in context to the game.

2 - The rating of the puzzle is displayed, so the user knows what level they are working at.

3 - A button that allows the user to return to the main menu.

Selecting a Game to Review



1 - A drop down menu that allows the user to select the game they want to review.

2 - A button that takes the user to the “Filter” menu where the user can decide which games are visible in the drop-down menu.

3 - A button that takes the user to the “Game Review” menu, allowing the user to review the game selected.

Filtering Games

1 - A drop-down menu that allows the user to decide the games visible by the colour the user was playing as.

2 - A drop-down menu that allows the user to decide the games visible by the result of the game with respect to the user.

3 - A drop-down menu allows the user to decide the games visible by whether the game was against the AI or not.

4 - A button that sets the filters made by the user updating the games displayed in the “Game Selector” menu.

Reviewing Games

1 - The Move Log is updated to display the move made in chess notation.

2 - A button that allows the user to make the next move that was made in the game.

3 - A button that allows the user to undo the move that was made in the game.

4 – A selector that allows the user to select who the user want to finish the game against.

5 - A button that allows the user to continue the game at the point where they were reviewing the game at.

6 – A button that allows the user to return to the main menu.

Security

Each user creates their own account and so the user can only log into the profiles which they know the username and password to. Every user’s password must be at least 8 characters long and must contain at least 1 special character. The plaintext password is never transmitted to the server. Instead, when creating the user, it uses a salt, which is a randomly generated key, to hash the plaintext. The hashed password and the salt are then stored on the server. When the user wants to log into their profile, the salt for the user is fetched from the server. The salt is then hashed with the user inputted password. Then if the hashed passwords are the same, the user is authenticated.

Data Integrity

There is validation which is conducted on the data as it passes between the server and the client. To ensure data integrity I have used the following:

If the user tries to create an account where the username is already being used, the server will return an error message which will then be returned to the user, allowing the user to change the username or log into the account belonging to that username.

If the user tries to save a game when the game name is already being used an error message will be displayed so the user can rename the name.

I made sure that the data that is saved to the server is correct and any other errors are caught like errors in the data type of the data, and a suitable error message is displayed to the user.

Technical Solution

Resources

File name	Image
bB.png	
bK.png	
bN.png	
bP.png	
bQ.png	
bR.png	
wB.png	
wK.png	
wN.png	
wP.png	
wQ.png	
wR.png	

Technical skills checklist

Skill	Location
Use of database	Server-side code and database section
Hash table	Database section – “user” table
Linked Lists	Client-side code – “Node” class
Stacks	“move_log”, “move_record” and “en_passant” in “Board” class (board.py lines 18,21,23)
Candidate written classes (OOP)	Throughout client-side code
Composition	Instantiation for the classes: “Chess” (main.py line 26) “Board” (chess.py line 9) “Piece” (board.py line 100)
Inheritance	piece.py: “Rook”, “Bishop”, “Knight”, “Queen”, “King”, “Pawn” classes inherit from the “Piece” class
User interface	Client-side code – “Main” class
Local variables	Throughout client-side code
Use of dictionaries	Throughout client-side code
Cross-table SQL statements	Server-side code lines 54-64
Aggregate SQL statements	Server-side code lines 54-64
Linked List Maintenance	Client-side code – “Stack” class
List Operations	Client-side code – “Node” class
Stack Operations	Client-side code - “Stack” class
Hashing	Client-side code – Creating an account (main.py lines 278 - 322)
Recursive Algorithms	Minimax Algorithm (engine.py lines 12 – 59)
User Defined Algorithms	Iterative Deepening (engine.py lines 61 – 70) Generating Legal Moves (board.py 264 -381) Converting Moves in Chess Notation (board.py lines 488 – 526)
Calling web API	Client-side code – “ChessClient” class
Parsing JSON	Client-side code – “ChessClient” class
File paths parameterised	piece.py (line 16)
Use of constants	Board.py (line 5)
Exception handling	Server-side code (lines 8,16)
Defensive programming	Throughout client-side code.

File Structure

```
C:.
|   board.py
|   chess.py
|   chessclient.py
|   engine.py
|   main.py
|   mouse.py
|   piece.py
|   stack.py
|
+---Pieces
|       bB.png
|       bK.png
|       bN.png
|       bP.png
|       bQ.png
|       bR.png
|       wB.png
|       wK.png
|       wN.png
|       wP.png
|       wQ.png
|       wR.png
```

Database

```
mysql> describe game;
+----------+--------------+------+-----+---------+----------------+
| Field    | Type         | Null | Key | Default | Extra          |
+----------+--------------+------+-----+---------+----------------+
| gameID   | int          | NO   | PRI | NULL    | auto_increment |
| userID   | int          | YES  |     | NULL    |                |
| name     | varchar(255) | YES  |     | NULL    |                |
| moves    | text         | YES  |     | NULL    |                |
| result   | char(1)      | YES  |     | NULL    |                |
| playedas | char(1)      | YES  |     | NULL    |                |
| AI       | tinyint(1)   | YES  |     | NULL    |                |
+----------+--------------+------+-----+---------+----------------+
7 rows in set (0.01 sec)

mysql> describe puzzle;
+----------+--------------+------+-----+---------+----------------+
| Field    | Type         | Null | Key | Default | Extra          |
+----------+--------------+------+-----+---------+----------------+
| puzzleID | int          | NO   | PRI | NULL    | auto_increment |
| FEN      | varchar(100) | YES  |     | NULL    |                |
| moves    | varchar(30)  | YES  |     | NULL    |                |
| rating   | int          | YES  |     | NULL    |                |
+----------+--------------+------+-----+---------+----------------+
4 rows in set (0.01 sec)

mysql> describe puzzleuser;
+----------+------+------+-----+---------+-------+
| Field    | Type | Null | Key | Default | Extra |
+----------+------+------+-----+---------+-------+
| puzzleID | int  | NO   | PRI | NULL    |       |
| userID   | int  | NO   | PRI | NULL    |       |
+----------+------+------+-----+---------+-------+
2 rows in set (0.01 sec)

mysql> describe user;
+-----------------+--------------+------+-----+---------+----------------+
| Field           | Type         | Null | Key | Default | Extra          |
+-----------------+--------------+------+-----+---------+----------------+
| userID          | int          | NO   | PRI | NULL    | auto_increment |
| username        | varchar(255) | NO   | UNI | NULL    |                |
| salt            | varchar(255) | NO   |     | NULL    |                |
| hashed_password | varchar(255) | NO   |     | NULL    |                |
+-----------------+--------------+------+-----+---------+----------------+
4 rows in set (0.00 sec)
```

Server-side code

```
001     from flask import Flask, request, jsonify
002     import MySQLdb
003     import traceback
004     app = Flask(__name__)
005     app.config['DEBUG'] = True
006
007     def connect_to_db():
008         try:
009             db = MySQLdb.connect("17mlee.mysql.eu.pythonanywhere-services.com", "17mlee", "qwerty132",
"17mlee$chess")
010             return db
011         except Exception:
012             return traceback.format_exc()
013
014     @app.route('/users',methods = ["POST"])
015     def create_user():
016         try:
017             data = request.get_json()
018             username = data["username"]
019             salt = data["salt"]
020             hashed_password = data["hashed_password"]
021             db = connect_to_db()
022             cursor = db.cursor()
023             cursor.execute("INSERT INTO user(username, salt, hashed_password) VALUES(%s, %s, %s)", (username,
salt, hashed_password))
024             db.commit()
025             cursor.close()
026             db.close()
027             return jsonify({"error": ""}), 400
028         except MySQLdb.IntegrityError:
029             return jsonify({"error": "Username already exists"}), 400
030
031     @app.route('/users/validate',methods = ["GET"])
032     def validate_user():
033         username = request.args.get('username')
034         hashed_password = request.args.get('hashed_password')
035         db = connect_to_db()
036         cursor = db.cursor()
037         cursor.execute("SELECT userID from user where username = %s and hashed_password = %s",
(username,hashed_password))
038         user = cursor.fetchall()
039         cursor.close()
040         db.close()
041         return jsonify(user)
042
043     @app.route('/users/salt',methods = ["GET"])
044     def get_user_salt():
045         username = request.args.get('username')
046         db = connect_to_db()
047         cursor = db.cursor()
048         cursor.execute("SELECT salt from user where username = %s", (username,))
049         salt = cursor.fetchall()
050         cursor.close()
051         db.close()
052         return jsonify(salt)
053
054     @app.route('/puzzles',methods = ["GET"])
055     def get_puzzles():
056         rating_min = request.args.get('rating_min', default = 0, type = int)
057         rating_max = request.args.get('rating_max', default = 100, type = int)
058         userID = request.args.get("userID")
059         db = connect_to_db()
060         cursor = db.cursor()
061         cursor.execute(f"SELECT * FROM puzzle WHERE rating = (SELECT MIN(rating) FROM puzzle WHERE rating < %s AND
rating > %s AND puzzleID NOT IN (SELECT puzzleID FROM puzzleuser WHERE userID = %s)) AND puzzleID NOT IN (SELECT
puzzleID FROM puzzleuser WHERE userID = %s) limit 1",(rating_max,rating_min,userID,userID))
062         puzzles = cursor.fetchall()
063         cursor.close()
064         db.close()
065         return jsonify(puzzles)
066
067     @app.route('/puzzles',methods = ["POST"])
068     def puzzle_completed():
069         data = request.get_json()
070         puzzleID = data["puzzleID"]
```

```
        userID = data["userID"]
        db = connect_to_db()
        cursor = db.cursor()
        cursor.execute(f"INSERT INTO puzzleuser(puzzleID, userID) VALUES(%s, %s)", (puzzleID, userID))
        db.commit()
        cursor.close()
        db.close()

    @app.route('/games',methods = ["GET"])
    def get_game():
        gameID = request.args.get('gameID')
        db = connect_to_db()
        cursor = db.cursor()
        cursor.execute(f"SELECT moves,result,playedas,AI from game where gameID = {gameID}")
        games = cursor.fetchall()
        cursor.close()
        db.close()
        return jsonify(games)

    @app.route('/games/names',methods = ["GET"])
    def get_game_names():

        userID = request.args.get('userID')
        result = tuple(request.args.getlist('result'))
        playedas = tuple(request.args.getlist('playedas'))
        AI = tuple(request.args.getlist("AI"))
        db = connect_to_db()
        cursor = db.cursor()
        # Generate placeholders for the IN clause
        result_placeholders = ', '.join(['%s'] * len(result))
        playedas_placeholders = ', '.join(['%s'] * len(playedas))
        AI_placeholders = ', '.join(['%s'] * len(AI))
        # Flatten the parameters into a tuple
        params = (userID, *result, *playedas, *AI)
        cursor.execute(f"SELECT name, gameID FROM game WHERE userID = %s AND result IN ({result_placeholders}) AND
playedas IN ({playedas_placeholders}) AND AI IN ({AI_placeholders})", params)
        games = cursor.fetchall()
        cursor.close()
        db.close()
        return jsonify(games)


    @app.route('/games',methods = ["POST"])
    def save_games():
        data = request.get_json()
        userID = data['userID']
        name = data["name"]
        AI = data['AI']
        moves = data['moves']
        result = data["result"]
        playedas = data["playedas"]
        db = connect_to_db()
        cursor = db.cursor()
        cursor.execute(f"INSERT INTO game(userID,name,moves,result, playedas,AI) VALUES(%s, %s,%s,%s, %s, %s)",
(userID,name, moves,result, playedas, AI))
        db.commit()
        cursor.close()
        db.close()

    if __name__ == '__main__':
        app.run(debug=True)
```

Client-side code

main.py

```
001     import pygame
002     from chess import Chess
003     from mouse import Mouse
004     import pygame_menu
005     from chessclient import ChessClient
006     import random
007     import bcrypt
008
009     pygame.init()
010     class Main():
011         def __init__(self):
012             self.__mouse = Mouse()
013             self.__screen_width = 1400
014             self.__screen_height = 800
015             self.__screen = pygame.display.set_mode((self.__screen_width,self.__screen_height)) # displaying the
screen with the dimensions
016             self.__running = True
017             self.__dragging = False
018             self.__board_dimension = self.__screen_height if self.__screen_width > self.__screen_height else
self.__screen_width
019             self.__screen_to_show= "Start"
020             self.__AI_diff = -1
021             self.__player1_colour = "w"
022             self.__mode = "PvP"
023             self.__fen = 'rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1'
024             self.__user = 0
025             self.__base_menu = "Start"
026             self.__chess = Chess(self.__screen,self.__mouse,self.__board_dimension)
027             self.__menus = {"Start" :
pygame_menu.Menu("Start",self.__screen_width,self.__screen_height,theme=pygame_menu.themes.THEME_BLUE),
028                             "Register":
pygame_menu.Menu("Register",self.__screen_width,self.__screen_height,theme=pygame_menu.themes.THEME_BLUE),
029                             "Main Menu" : pygame_menu.Menu("Main
Menu",self.__screen_width,self.__screen_height,theme=pygame_menu.themes.THEME_BLUE),
030                             "Login" :
pygame_menu.Menu("Login",self.__screen_width,self.__screen_height,theme=pygame_menu.themes.THEME_BLUE),
031                             "Puzzle Selector" : pygame_menu.Menu("Puzzle
Selector",self.__screen_width,self.__screen_height,theme=pygame_menu.themes.THEME_BLUE),
032                             "Difficulty" :
pygame_menu.Menu("Difficulty",self.__screen_width,self.__screen_height,theme=pygame_menu.themes.THEME_BLUE),
033                             "Game" : pygame_menu.Menu("Game",self.__screen_width -
self.__board_dimension,self.__screen_height,theme=pygame_menu.themes.THEME_BLUE),
034                             "Game Selector" : pygame_menu.Menu("Game
Selector",self.__screen_width,self.__screen_height,theme=pygame_menu.themes.THEME_BLUE),
035                             "Puzzles" : pygame_menu.Menu("Puzzles",self.__screen_width -
self.__board_dimension,self.__screen_height,theme=pygame_menu.themes.THEME_BLUE),
036                             "Filter" :
pygame_menu.Menu("Filter",self.__screen_width,self.__screen_height,theme=pygame_menu.themes.THEME_BLUE),
037                             "Game Review" : pygame_menu.Menu("Game Review",self.__screen_width -
self.__board_dimension,self.__screen_height,theme=pygame_menu.themes.THEME_BLUE)}
038
039         def GameLoop(self): # the infinite loop where window is run from
040             self.create_menus()
041             set_up = False
042             while self.__running==True:
043                 if self.__screen_to_show not in ["Game","Puzzles","Game Review"]:
044                     set_up = False
045                     self.__menus[self.__screen_to_show].mainloop(self.__screen)
046                 else:
047                     if set_up==False: # setting up the board for the game
048                         self.__menus[self.__screen_to_show].enable()
049                         if self.__screen_to_show == "Game" or self.__screen_to_show == "Game Review":
050                             self.__fen = 'rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1'
051                         elif self.__screen_to_show == "Puzzles":
052                             self.__menus[self.__screen_to_show].get_widget("accuracy").set_title("")
053                             self.__menus[self.__screen_to_show].get_widget("rating").set_title("Rating: " +
str(self.__puzzle[3]) )
054                             number_of_puzzle_moves = 0
055                             if "w" in self.__fen:
056                                 self.__player1_colour = "b"
057                             else:
```

```
058                                 self.__player1_colour = "w"
059                             self.__chess = Chess(self.__screen,self.__mouse,self.__board_dimension)
060 
061                             self.__chess.set_up_playing_condition(self.__player1_colour,self.__AI_diff,self.__mode,sel
f.__fen)
062                             self.update_move_record_label()
063                             set_up = True
064 
065                         if self.__screen_to_show == "Puzzles":
066                             if number_of_puzzle_moves % 2 == 0:
067                                 puzzle_moves = self.__puzzle[2].split()
068 
069                                 start_pos, end_pos =
self.__chess.cal_start_end_position(puzzle_moves[number_of_puzzle_moves])
070                                 self.__chess.non_player_move(start_pos,end_pos)
071                                 number_of_puzzle_moves += 1
072                         self.__chess.display_chess(self.__dragging)
073 
074                     events = pygame.event.get()
075                     if self.__menus[self.__screen_to_show].is_enabled():
076                         self.__menus[self.__screen_to_show].update(events)
077                         self.__menus[self.__screen_to_show].draw(self.__screen)
078                     for event in events:
079                         if event.type == pygame.QUIT:
080                             self.__running = False
081 
082                         if event.type == pygame.MOUSEBUTTONDOWN and event.button == 1:
083                             self.__dragging = True
084                             self.__mouse.update_initial_position()
085                             if self.__screen_to_show == "Game":
086                                 self.__chess.select_piece()
087                             elif self.__screen_to_show == "Puzzles":
088                                 self.__chess.select_piece_in_puzzle()
089 
090                         if event.type == pygame.MOUSEMOTION and self.__dragging and (self.__screen_to_show == "Game"
or self.__screen_to_show == "Puzzles"):
091                             self.__mouse.drag(event.rel) ## allowing for dragging of pieces
092 
093                         if event.type == pygame.MOUSEBUTTONUP and event.button == 1:
094 
095                             if self.__screen_to_show == "Game":
096                                 self.__dragging = False
097                                 self.__mouse.update_final_position()
098 
099                                 self.__chess.piece_moved()
100                                 self.__mouse.reset()
101                                 self.__chess.display_chess(self.__dragging)
102                                 self.update_move_record_label()
103 
104                                 pygame.display.update()
105                                 self.__chess.AI_move()
106                                 self.update_move_record_label()
107                                 self.__chess.display_chess(self.__dragging)
108 
109                             elif self.__screen_to_show == "Puzzles":
110                                 self.__dragging = False
111                                 self.__mouse.update_final_position()
112                                 start_pos, end_pos =
self.__chess.cal_start_end_position(puzzle_moves[number_of_puzzle_moves])
113                                 if self.__chess.puzzle_piece_moved((start_pos,end_pos)) == True:
114                                     number_of_puzzle_moves += 1
115                                     self.__menus[self.__screen_to_show].get_widget("accuracy").set_title("Correct")
116                                 else:
117                                     self.__menus[self.__screen_to_show].get_widget("accuracy").set_title("Try Again")
118                                 if number_of_puzzle_moves == len(puzzle_moves):
119                                     self.__chess.display_chess(self.__dragging)
120                                     pygame.display.update()
121                                     ChessClient.puzzle_completed(self.__user,self.__puzzle[0])
122                                     self.get_puzzle()
123                                     if len(self.__puzzle) >0:
124                                         set_up = False
125                                         self.__menus[self.__screen_to_show].get_widget("rating").set_title("Rating: "
+ str(self.__puzzle[3]) )
126                                         self.__menus[self.__screen_to_show].get_widget("accuracy").set_title("")
127                                 self.__mouse.reset()
128                     pygame.display.update()
```

```
129         def update_screen(self,screen): # update menu to show
130             if screen not in ["Game", "Puzzles","Game Review"]:
131                 self.__menus[self.__base_menu].enable()
132                 self.__menus[self.__base_menu]._open(self.__menus[screen])
133                 self.__menus["Game"].get_widget("Move Record").set_title("")
134             else:
135                 self.__menus[self.__screen_to_show].disable()
136                 self.__menus[screen].enable()
137             self.__screen_to_show = screen # previous menu determines what mode to play
138             if self.__screen_to_show == "Puzzle Selector":
139                 self.__mode = "Puzzles"
140             elif self.__screen_to_show == "Difficulty":
141                 self.__mode = "AI"
142             elif self.__screen_to_show =="Main Menu":
143                 self.__AI_diff = -1
144                 self.__mode = "PvP"
145             elif self.__screen_to_show == "Game Selector":
146                 self.get_games_name()
147 
148         def create_menus(self): # creating the widgets for the menus
149             self.__menus["Start"].add.label("A Chess Game",font_size=50,font_color=(0,0,0))
150             self.__menus["Start"].add.button('Login', self.update_screen,"Login")
151             self.__menus["Start"].add.button('Register', self.update_screen,"Register")
152 
153             self.__menus["Register"].add.text_input('Username :',textinput_id='username')
154             self.__menus["Register"].add.text_input('Password :', password=True,textinput_id='password')
155             self.__menus["Register"].add.label('', label_id="error")
156             self.__menus["Register"].add.button('Submit', self.submit_login)
157 
158             self.__menus["Login"].add.text_input('Username :',textinput_id='username')
159             self.__menus["Login"].add.text_input('Password :', password=True,textinput_id='password')
160             self.__menus["Login"].add.label('', label_id="error")
161             self.__menus["Login"].add.button('Submit', self.submit_login)
162 
163             self.__menus["Puzzles"].set_absolute_position(self.__board_dimension,0)
164             self.__menus["Puzzles"].add.label("",label_id="rating")
165             self.__menus["Puzzles"].add.label("",label_id="accuracy")
166             self.__menus["Puzzles"].add.button('Back', self.update_screen, "Main Menu")
167 
168             self.__menus["Main Menu"].add.button('Play Against AI', self.update_screen,"Difficulty")
169             self.__menus["Main Menu"].add.button('Play Against Human', self.update_screen,"Game")
170             self.__menus["Main Menu"].add.button('Puzzles', self.update_screen,"Puzzle Selector")
171             self.__menus["Main Menu"].add.button('Game Selector', self.update_screen,"Game Selector")
172             self.__menus["Main Menu"].add.button('Quit', pygame_menu.events.EXIT)
173 
174             self.__menus["Filter"].add.dropselect('Colour You Played As:', [('Black',
("b")),('White',("w")),("Both",("w","b"))],dropselect_id="PlayedAs",default=2)
175             self.__menus["Filter"].add.dropselect_multiple('Result:',[('Win', ("W")), ('Loss',
("L")),('Draw',("D")),("Undecided",("U"))],dropselect_multiple_id="Result",default=[0,1,2])
176             self.__menus["Filter"].add.dropselect('Game Against:',[('AI', (1)), ('Human',
(0)),('Both',(0,1))],dropselect_id="AI",default=2)
177             self.__menus["Filter"].add.button('Set Filter', self.update_screen,"Game Selector")
178 
179             self.__menus["Game Selector"].add.dropselect('Games:',["TempValue"],selection_box_width =
800,dropselect_id="Game Selector")
180             self.__menus["Game Selector"].add.button("Filter",self.update_screen,"Filter")
181             self.__menus["Game Selector"].add.button('Load Game', self.load_game)
182             self.__menus["Game Selector"].add.label("",label_id = "Error")
183 
184             self.__menus["Difficulty"].add.selector('Game Mode :', [('Easy',
2),('Intermediate',3),('Hard',4)],selector_id="Game Mode")
185             self.__menus["Difficulty"].add.selector('Colour :', [("White", "w"), ('Black', "b"),('Random',"r")],
selector_id= "Colour")
186             self.__menus["Difficulty"].add.button('Play', self.settings_against_AI)
187 
188             self.__menus["Game"].set_absolute_position(self.__board_dimension,0)
189             self.__menus["Game"].add.label("Move Log")
190             self.__menus["Game"].add.label("",font_size = 15, wordwrap=True,label_id="Move Record")
191             self.__menus["Game"].add.label("",label_id="Game State")
192             self.__menus["Game"].add.button('Undo',self.undo_move)
193             self.__menus["Game"].add.button('Back', self.update_screen, "Main Menu")
194             self.__menus["Game"].add.text_input('Name :',textinput_id='Name')
195             self.__menus["Game"].add.button('Save', self.save_game)
196             self.__menus["Game"].add.label("",label_id="Error")
197 
198             self.__menus["Game Review"].set_absolute_position(self.__board_dimension,0)
199             self.__menus["Game Review"].add.label("Move Log")
```

```
200             self.__menus["Game Review"].add.label("",font_size = 15, wordwrap=True,label_id="Move Record")
201             self.__menus["Game Review"].add.label("",label_id="Game State")
202             self.__menus["Game Review"].add.button('Make Move',self.make_move)
203             self.__menus["Game Review"].add.button('Undo Move',self.undo_move)
204             self.__menus["Game Review"].add.label("Continue Playing Against:")
205             self.__menus["Game Review"].add.selector('', [("No AI", 0), ('Easy',
206 2),('Intermediate',3),('Hard',4)], selector_id= "AI")
207             self.__menus["Game Review"].add.button('Continue Game',self.continue_game)
208             self.__menus["Game Review"].add.button('Back', self.update_screen, "Main Menu")
209 
210             self.__menus["Puzzle Selector"].add.range_slider("Rating",(0,2000),(0,2000),100, rangeslider_id=
211 "Rating")
212             self.__menus["Puzzle Selector"].add.label("",label_id = "Error")
213             self.__menus["Puzzle Selector"].add.button('Play',self.get_puzzle)
214 
215         def get_games_name(self): # get game names from server
216             input_data = self.__menus["Filter"].get_input_data()
217             playedas = input_data['PlayedAs'][0][1]
218             result = tuple(t[1] for t in input_data['Result'][0])
219             AI = input_data['AI'][0][1]
220             if result != ():
221                 self.__menus["Game Selector"].get_widget("Game
222 Selector").update_items(ChessClient.get_games_name(self.__user,result,playedas,AI))
223             else:
224                 self.__menus["Game Selector"].get_widget("Game Selector").update_items([])
225 
226         def settings_against_AI(self):
227             input_data = self.__menus["Difficulty"].get_input_data()
228             self.__AI_diff = input_data["Game Mode"][0][1]
229             colour = input_data["Colour"][0][1]
230             if colour == "r":
231                 colour = random.choice(["w","b"])
232             self.__player1_colour = colour
233             self.update_screen("Difficulty")
234             self.update_screen("Game")
235 
236         def get_puzzle(self):
237             input_data = self.__menus["Puzzle Selector"].get_input_data()
238             rating = input_data['Rating']
239             min_rating = round(rating[0])
240             max_rating = round(rating[1])
241             puzzle = ChessClient.get_puzzle(self.__user, min_rating,max_rating)
242             if len(puzzle) > 0:
243                 self.__puzzle = puzzle[0]
244                 self.__fen = self.__puzzle[1]
245                 self.update_screen("Puzzle Selector")
246                 self.__menus["Puzzle Selector"].get_widget("Error").set_title("")
247                 self.update_screen("Puzzles")
248             else:
249                 self.__puzzle = []
250                 self.update_screen("Main Menu")
251                 self.update_screen("Puzzle Selector")
252                 self.__menus["Puzzle Selector"].get_widget("Error").set_title("There are no puzzles within the
253 selected range")
254 
255         def continue_game(self): # continue the game from position in game review
256             input_data = self.__menus["Game Review"].get_input_data()
257             AI = input_data["AI"][0][1]
258             if AI != 0:
259                 self.__chess.add_engine(AI)
260             self.update_screen("Game")
261 
262 
263         def make_move(self): # used for the game review screen  - makes the move that was stored on the database
264             move_to_make = self.__chess.get_move_record().length()
265             if move_to_make < len(self.__moves):
266                 start_pos, end_pos = self.__moves[move_to_make].split(",")
267                 start_pos, end_pos = int(start_pos), int(end_pos)
268                 if move_to_make % 2 ==0:
269                     self.__chess.change_colour_to_move("w")
270                 else:
271                     self.__chess.change_colour_to_move("b")
272                 self.__chess.non_player_move(start_pos,end_pos)
273                 self.update_move_record_label()
274 
275         def undo_move(self):
276             if self.__AI_diff != -1:
```

```
273                 self.__chess.undo_move()
274             self.__chess.select_piece()
275             self.__chess.undo_move()
276             self.__chess.select_piece()
277 
278         def submit_login(self): # Registering/Logging In to account
279             if self.__screen_to_show == "Register":
280                 input_data = self.__menus["Register"].get_input_data()
281                 username = input_data["username"]
282                 password = input_data["password"]
283                 if len(password) < 8:
284                     self.__menus["Register"].get_widget("error").set_title("Password needs to be at least 8
characters long")
285                 else:
286                     count = 0
287                     for character in password:
288                         if character in " !#$%&'()*+,-./:;<=>?@[\]^_`{|}~":
289                             count += 1
290                     if count == 0:
291                         self.__menus["Register"].get_widget("error").set_title("Password needs to contain at least
1 special character")
292                     else: # producing the hash value for the password
293                         salt = bcrypt.gensalt()
294                         hashed_password = bcrypt.hashpw(password.encode("utf-8"),salt)
295                         response = ChessClient.create_user(username,salt.decode("utf-
8"),hashed_password.decode("utf-8")) #convert back to string and send to server
296                         if response["error"] == "Username already exists":
297                             self.__menus["Register"].get_widget("error").set_title("Username already exists")
298                         else:
299                             response = ChessClient.validate_user(username,hashed_password.decode("utf-8"))
300                             self.__user = response[0][0]
301                             self.__menus["Register"].get_widget("error").set_title("Inputted user details are
valid")
302 
303             elif self.__screen_to_show == "Login":
304 
305                 input_data = self.__menus["Login"].get_input_data()
306                 username = input_data["username"]
307                 password = input_data["password"].encode("utf-8")
308                 response = ChessClient.get_salt(username)
309                 if not response:
310                     self.__menus["Login"].get_widget("error").set_title("Username does not exist")
311                 else: # checks if the hash values are the same
312                     salt = response[0][0].encode("utf-8")
313                     hashed_password = bcrypt.hashpw(password,salt)
314                     response = ChessClient.validate_user(username,hashed_password.decode("utf-8"))
315                     if not response:
316                         self.__menus["Login"].get_widget("error").set_title("Incorrect Password")
317                     else:
318                         self.__user = response[0][0]
319                         self.__menus["Login"].get_widget("error").set_title("")
320                         self.__menus[self.__base_menu].disable()
321                         self.__base_menu = "Main Menu"
322                         self.update_screen("Main Menu")
323 
324         def load_game(self):
325             input_data = self.__menus["Game Selector"].get_input_data()
326             if input_data != {}:
327                 gameID = input_data["Game Selector"][0][1]
328                 game = ChessClient.get_game_moves(gameID)[0]
329                 self.__player1_colour = game[2]
330                 self.__moves = game[0].split()
331                 self.__menus["Game Selector"].get_widget("Error").set_title("")
332                 self.update_screen("Game Review")
333             else:
334                 self.__menus["Game Selector"].get_widget("Error").set_title("Select a Game")
335 
336         def save_game(self):
337             move_log = self.__chess.get_move_log()
338             moves = []
339             while move_log.length() > 0:
340                 moves += [move_log.pop()]
341             moves = list(reversed(moves)) # gets start square and end square for all moves in order
342             list_of_moves = [(sublist[0], sublist[1]) for sublist in moves if len(sublist) >= 2]
343             list_of_moves_str = ' '.join([f'{x},{y}' for x, y in list_of_moves]) # store as a string so it can be
stored in the database
344             result = self.__chess.get_result()
```

```
345             playedas = self.__player1_colour
346             AI = self.__chess.playingAI()
347             name = self.__menus["Game"].get_input_data()["Name"]
348             all_user_games = ChessClient.get_games_name(self.__user, ("W","L","D","U"), ("w","b"), (0,1))
349             all_user_game_names = [games[0] for games in all_user_games]
350             if len(name) > 0 and name not in all_user_game_names:
351                 ChessClient.save_game(self.__user,name,list_of_moves_str,result,playedas,AI)
352                 self.__menus["Game"].get_widget("Error").set_title("")
353                 self.update_screen("Main Menu")
354             else:
355                 self.__menus["Game"].get_widget("Error").set_title("Enter a valid name")
356 
357         def update_move_record_label(self):
358             if self.__screen_to_show == "Game" or self.__screen_to_show == "Game Review":
359                 self.__menus[self.__screen_to_show].get_widget("Move
Record").set_title(self.__chess.return_move_record_as_string())
360                 self.__menus[self.__screen_to_show].get_widget("Game
State").set_title(self.__chess.state_of_game())
361 
362     game1= Main()
363     game1.GameLoop()
```

chess.py

```
001     from stack import Stack
002     import pygame
003
004     class Chess():
005         def __init__(self,screen,mouse,board_dimension):
006             self.__mouse = mouse
007             self.__screen = screen
008             self.__board = Board(board_dimension,self.__mouse)
009             self.__selected_piece_position = 100
010             self.__player1_colour = None
011             self.__player2_colour = None
012             self.__AI = None
013             self.__mode = None
014             self.__colour_to_move = None
015
016         def set_up_playing_condition(self,player1,AI_difficulty,mode,fen): # sets up game conditions for the given
mode
017             self.__mode = mode
018             if player1 == "w":
019                 self.__player1_colour = "w"
020                 self.__player2_colour = "b"
021             else:
022                 self.__player1_colour = "b"
023                 self.__player2_colour = "w"
024             if self.__mode == "Puzzles":
025                 self.__colour_to_move = self.__player1_colour
026             else:
027                 self.__colour_to_move = "w" if "w" in fen else "b"
028             self.__board.create_board_set_up(self.__player1_colour,fen)
029             self.__board.display_board(self.__screen)
030             self.__board.display_pieces(self.__screen)
031             if self.__mode == "AI":
032                 pygame.display.update()
033                 self.add_engine(AI_difficulty)
034
035         def add_engine(self,AI_difficulty): # makes the engine the 2nd player
036             self.__AI = Engine(self.__board,self.__player2_colour,AI_difficulty)
037             if self.__player1_colour == "b":
038                 self.AI_move()
039
040         def select_piece(self): # allows the user to select a piece in game
041             if self.__board.return_move_log().length()%2 == 0:
042                 self.__colour_to_move = "w"
043             else:
044                 self.__colour_to_move = "b"
045             if not(self.__board.check_if_draw_by_repetition()): # prevents users from being able to move when its
a draw
046                 self.__board.all_legal_moves(self.__colour_to_move)
047                 self.__selected_piece_position = self.__board.piece_picked_up(self.__colour_to_move)
048
049         def select_piece_in_puzzle(self): # allows the user to select a piece in a puzzle
050             self.__colour_to_move = "w" if self.__colour_to_move == "b" else "b"
051             self.__board.all_legal_moves(self.__colour_to_move)
052             self.__selected_piece_position = self.__board.piece_picked_up(self.__colour_to_move)
053
054         def move_piece(self, end_position, selected_piece_position): # valiades the moves made by the user
055             if 0 <= selected_piece_position < 64:
056                 if 0 <= end_position[0] < 8 and 0 <= end_position[1] < 8:
057                     position = end_position[0] * 8 + end_position[1]
058                     if self.__board.validate_user_move((selected_piece_position, position)):
059                         self.__board.move((selected_piece_position, position))
060                         piece = self.__board.return_piece(position)
061                         self.__board.update_piece_rect(piece, position)
062                         self.__board.convert_move_log_to_notation()
063                         self.__board.update_rects()
064                         return True
065             return False
066
067         def update_piece_rect_if_needed(self, selected_piece_position): #updates the position of the pieces on the
board
068             if selected_piece_position != 100:
069                 piece = self.__board.return_piece(selected_piece_position)
070                 if piece is not None:
071                     self.__board.update_piece_rect(piece, selected_piece_position)
072
073         def piece_moved(self):
```

```
074             end_position = self.__board.convert_mouse_end_position_to_piece_position()
075             moved = self.move_piece(end_position, self.__selected_piece_position)
076             if not moved:
077                 self.update_piece_rect_if_needed(self.__selected_piece_position)
078             return moved
079 
080         def puzzle_piece_moved(self, move): # checks if the move made by the user is the same the correct move
held in database
081             end_position_coord = self.__board.convert_mouse_end_position_to_piece_position()
082             if self.__selected_piece_position == move[0]:
083                 end_position = end_position_coord[0] * 8 + end_position_coord[1]
084                 if end_position == move[1]:
085                     moved = self.move_piece(end_position_coord, self.__selected_piece_position)
086                     self.update_piece_rect_if_needed(self.__selected_piece_position)
087                     return moved
088             self.update_piece_rect_if_needed(self.__selected_piece_position)
089             self.__colour_to_move = "w" if self.__colour_to_move == "b" else "b"
090             return False
091 
092         def undo_move(self):
093             self.__board.undo_move()
094             self.__board.update_rects()
095             self.__board.undo_move_record()
096 
097         def change_colour_to_move(self,colour):
098             self.__colour_to_move = colour
099 
100         def AI_move(self): # allows the AI to move
101             if self.__AI != None:
102 
103                 if self.__board.return_move_log().length()%2 == 0:
104                     colour_to_move = "w"
105                 else:
106                     colour_to_move = "b"
107                 self.__colour_to_move = colour_to_move
108                 if colour_to_move == self.__player2_colour:
109                     if self.__player2_colour == "w":
110                         maximising_player = True
111                     else:
112                         maximising_player = False
113                     self.__board.check_if_end_game()
114                     self.__AI.update_board(self.__board)
115                     best_move,evaluation = self.__AI.iterative_deepening(maximising_player)
116                     if best_move == None:
117                         return
118                     start_position = best_move[0]
119                     end_position = best_move[1]
120                     self.non_player_move(start_position,end_position)
121 
122         def non_player_move(self, start_position, end_position): # used for Puzzles/AI to move the opposition
pieces
123             self.__board.all_legal_moves(self.__colour_to_move)
124             self.__board.move((start_position,end_position))
125             self.__board.convert_move_log_to_notation()
126             self.__board.update_rects()
127             self.__colour_to_move = "w" if self.__colour_to_move == "b" else "b"
128 
129         def cal_start_end_position(self,move):
130             start_position = move[:2]
131             end_position = move[2:4]
132             start_position = self.__board.convert_puzzle_move_to_move(start_position)
133             end_position = self.__board.convert_puzzle_move_to_move(end_position)
134             return start_position,end_position
135 
136         def display_chess(self,dragging):
137             self.__board.display_board(self.__screen)
138             if 0 <=self.__selected_piece_position < 64:
139                 if dragging == True:
140                     self.__board.displaying_possible_moves(self.__screen,self.__selected_piece_position)
141             self.__board.display_pieces(self.__screen)
142 
143         def get_move_record(self):
144             return(self.__board.return_move_record())
145 
146         def get_move_log(self):
147             return(self.__board.return_move_log())
148 
```

```
        def get_result(self):
            move_record = self.__board.return_move_record()
            if move_record.length() > 0:
                last_move = move_record.peek()
                if "++" in last_move:
                    if move_record.peek() % 2 == 0 and self.__player1_colour == "w":
                        return "L"
                    elif move_record.peek() % 2 == 1 and self.__player1_colour == "b":
                        return "L"
                    else:
                        return "W"
                elif self.__board.check_if_stalemate() or self.__board.check_if_draw_by_repetition():
                    return "D"
            return "U"

        def playingAI(self):
            if self.__AI != None:
                return True
            else:
                return False

        def return_move_record_as_string(self):
            move_record = self.__board.return_move_record()
            move_record_string = ""
            temp_stack = Stack()
            while not move_record.is_empty(): # pop items from stack and push them onto a temporary stack
                item = move_record.pop()
                temp_stack.push(item)
            while not temp_stack.is_empty(): # pop items from temporary stack and push them back onto the original
stack and add them to a string
                move_record_string += str(temp_stack.peek()) + ", "
                move_record.push(temp_stack.pop())

            move_record_string = move_record_string.rstrip(", ")
            return move_record_string
        def state_of_game(self):
            move_record = self.__board.return_move_record()
            if move_record.length() > 0:
                last_move = move_record.peek()
                if "++" in last_move:
                    return "Checkmate"
                elif "+" in last_move:
                    return "Check"
                else:
                    if self.__board.check_if_stalemate():
                        return "Stalemate"
                    elif self.__board.check_if_draw_by_repetition():
                        return "Draw by repetition"
            return ""
```

board.py

```
001     import pygame
002     from piece import *
003     from mouse import Mouse
004     from stack import Stack
005     N,E,S,W = -8,1,8,-1
006     class Board():
007         def __init__(self,board_size,mouse):
008             self.__square_size = int(board_size/8)
009             self.__boardcolour1 = (240,217,181)
010             self.__boardcolour2 = (181,136,99)
011             self.__boardcolour3 = (205,92,92) # highlights legal moves
012             self.__boardcolour4 = (205,210,106) # highlights preview move
013             self.__boardcolour5 = (170,162,58)
014             self.__board = dict()
015             self.__mouse = mouse
016             self.__position_black_king = None
017             self.__position_white_king = None
018             self.__move_log = Stack()
019             self.__columns_to_files = {0:"a",1:"b",2:"c",3:"d",4:"e",5:"f",6:"g",7:"h"}
020             self.__rows_to_rank = {0:"8",1:"7",2:"6",3:"5",4:"4",5:"3",6:"2",7:"1"}
021             self.__move_record = Stack()
022             self.__castling_rights = []
023             self.__en_passant = Stack()
024             self.__legal_moves = []
025             self.__perspective = "w"
026             self.__piece_classes = {'p': Pawn, 'R': Rook, 'N': Knight, 'B': Bishop, 'Q': Queen, 'K': King}
027             self.__pieces_moves = {"R":[N,E,S,W],
028                                    "B":[N+E,S+E,S+W,N+W],
029                                    "Q":[N,E,S,W,N+E,S+E,S+W,N+W],
030                                    "K":[N,E,S,W,N+E,S+E,S+W,N+W],
031                                    "p":[N,N+E,N+W],
032                                    "N":[E+N+N,N+E+E,S+E+E,S+S+E,S+S+W,S+W+W,N+W+W,N+N+W]}
033             self.__pieces_moves_types = {"R":[[N],[E],[S],[W]],
034                                    "B":[[N,E],[S,E],[S,W],[N,W]],
035                                    "Q":[[N],[E],[S],[W],[N,E],[S,E],[S,W],[N,W]],
036                                    "K":[[N],[E],[S],[W],[N,E],[S,E],[S,W],[N,W]],
037                                    "p":[[N],[N,E],[N,W]],
038                                    "N":[[E,N,N],[N,E,E],[S,E,E],[S,S,E],[S,S,W],[S,W,W],[N,W,W],[N,N,W]]}
039             self.__white_pieces = []
040             self.__black_pieces = []
041             self.__pieces = {"w":self.__white_pieces,"b":self.__black_pieces}
042             self.__end_game = False
043 
044         def rotate_board(self): # switches all the variables so that the white/black pieces can be on either side
of board
045             self.__board = {63-i:self.__board[i] for i in range(len(self.__board))}
046             self.__perspective = "b"
047             temp_move_log = Stack()
048             temp_move_record = Stack()
049             temp_en_passant = Stack()
050             while not self.__move_log.is_empty(): # pop items off the stack, modify them, and push them onto the
temporary stack
051                 i, j, s, t = self.__move_log.pop()
052                 temp_move_log.push((63-i, 63-j, s, t))
053                 i, j = self.__move_record.pop()
054                 temp_move_record.push((63-i, 63-j))
055                 i = self.__en_passant.pop()
056                 if i != None:
057                     i = 63-i
058                 temp_en_passant.push(i)
059             while not temp_move_log.is_empty(): # pop items off the temporary stack and push them back onto the
original stack
060                 self.__move_log.push(temp_move_log.pop())
061                 self.__move_record.push(temp_move_record.pop())
062                 self.__en_passant.push(temp_en_passant.pop())
063             self.__legal_moves = [(63-i,63-j) for i,j in self.__legal_moves]
064             self.__mouse.update_selected_piece(None)
065             self.__white_pieces = [63-i for i in self.__white_pieces]
066             self.__black_pieces = [63-i for i in self.__black_pieces]
067             self.__position_white_king = 63-self.__position_white_king
068             self.__position_black_king = 63-self.__position_black_king
069             self.__pieces = {"w":self.__white_pieces,"b":self.__black_pieces}
070             for piece in self.__pieces["w"]:
071                 self.__board[piece].rotate_piece_square_table()
072             for piece in self.__pieces["b"]:
073                 self.__board[piece].rotate_piece_square_table()
```

```
074             self.__columns_to_files = {i: chr(104 - i) for i in range(8)}
075             self.__rows_to_rank = {i: str(i + 1) for i in range(8)}
076             self.update_rects()
077 
078         def create_board_set_up(self,perspective,fen): # Algorithm to Interpret FEN string
079             self.__perspective = perspective
080             square = 0
081             wr_pos = []
082             br_pos = []
083             for character in fen:
084                 if character == " ":
085                     index = fen.index(character)
086                     index += 1
087                     break
088                 if character == "/":
089                     pass
090                 elif character.isdigit():
091                     for i in range(int(character)):
092                         self.__board[square] = None
093                         square += 1
094                 else:
095                     if character.isupper():
096                         colour = "w"
097                         self.__white_pieces.append(square)
098                     else:
099                         colour = "b"
100                         self.__black_pieces.append(square)
101 
102                     if character == "p" or character == "P":
103                         piece_class = self.__piece_classes.get(character.lower())
104                     else:
105                         piece_class = self.__piece_classes.get(character.upper())
106                     if character == "K":
107                         self.__position_white_king = square
108                     if character == "k":
109                         self.__position_black_king = square
110                     if piece_class:
111                         row = square // 8
112                         column = square %8
113                         self.__board[square] = piece_class(colour,(column * self.__square_size, row *
self.__square_size),self.__square_size)
114                         if character == "R":
115                             wr_pos.append(square)
116                         if character == "r":
117                             br_pos.append(square)
118                     square += 1
119             index += 2
120             if fen[index] == "-": # this character in the fen string represents the castling rights
121                 for i in wr_pos:
122                     self.__board[i].add_move_made()
123                 for i in br_pos:
124                     self.__board[i].add_move_made()
125             else:
126                 s = index
127                 while True:
128                     if fen[index] == " ":
129                         break
130                     index +=1
131                 c_rights = set(fen[s:index])
132                 p_rights = set("KQkq")
133                 self.__castling_rights = list(p_rights.intersection(c_rights))
134 
135             if fen[index] == "-": # this character in the fen string represents the possible en passant square
136                 self.__en_passant.push(None)
137             else:
138                 column = next((k for k, v in self.__columns_to_files.items() if v == fen[index]),None)
139                 row = next((k for k, v in self.__rows_to_rank.items() if v == fen[index+1]),None)
140                 sq = None
141                 if column != None:
142                     sq = (8*row +column)
143                 self.__en_passant.push(sq)
144             if self.__perspective == "b":
145                 self.rotate_board()
146 
147         def display_board(self,screen):
148             for i in range(len(self.__board)):
149                 row = i // 8
```

```
150                 column = i % 8
151                 rect=(column *self.__square_size,row*self.__square_size,self.__square_size,self.__square_size)
152                 if (column+row)%2 == 0: # test whether the square is a light/dark square
153                     pygame.draw.rect(screen,self.__boardcolour1,rect)
154                 else:
155                     pygame.draw.rect(screen,self.__boardcolour2,rect)
156 
157             if self.__move_log.length() > 0:
158                 for i in range(2): # show the previous move made
159                     start_row = self.__move_log.peek()[i]//8
160                     start_column = self.__move_log.peek()[i]%8
161                     rect=(start_column
*self.__square_size,start_row*self.__square_size,self.__square_size,self.__square_size)
162                     if (start_column+start_row)%2 == 0:
163                         pygame.draw.rect(screen,self.__boardcolour4,rect)
164                     else:
165                         pygame.draw.rect(screen,self.__boardcolour5,rect)
166 
167         def display_pieces(self,screen):
168             selected_piece = self.__mouse.return_selected_piece()
169             for i in self.__white_pieces:
170                 piece = self.__board[i]
171                 if selected_piece != piece:
172                     piece.display_piece(screen)
173             for i in self.__black_pieces:
174                 piece = self.__board[i]
175                 if selected_piece != piece:
176                     piece.display_piece(screen)
177             if selected_piece != None:
178                 selected_piece.display_piece(screen)
179 
180         def piece_picked_up(self,colour_to_move):
181             x_prev,y_prev = self.__mouse.return_initial_position()
182             prev_column = x_prev// self.__square_size
183             prev_row = y_prev// self.__square_size
184             if 0 <= prev_row <8 and 0 <= prev_column <8:
185                 square = prev_row * 8 +prev_column
186                 piece = self.__board[square]
187                 if piece != None:
188                     if piece.return_colour() == colour_to_move:
189                         self.__mouse.update_selected_piece(piece)
190                         return square
191             return 100
192 
193         def convert_mouse_end_position_to_piece_position(self):
194             x_new,y_new = self.__mouse.return_final_position()
195             new_column = x_new// self.__square_size
196             new_row = y_new//self.__square_size
197             return (new_row,new_column)
198 
199         def update_piece_rect(self,piece,piece_position):
200             row = piece_position // 8
201             column = piece_position % 8
202             rect =  (column*self.__square_size,row*self.__square_size)
203             piece.update_rect(rect)
204 
205         def update_board(self,piece,piece_position,end_position):
206             self.__board[piece_position] = None
207             self.__board[end_position] = piece
208             colour = piece.return_colour()
209             if end_position != piece_position:
210                 self.__pieces[colour].remove(piece_position)
211                 self.__pieces[colour].append(end_position)
212             elif piece_position == end_position:
213                 self.__pieces[colour].append(end_position)
214 
215         def move(self,move):
216             piece_position = move[0]
217             end_position = move[1]
218             special_case = ""
219             en_passant = None
220             taken_piece = None
221             piece = self.__board[piece_position]
222             colour = piece.return_colour()
223             op_colour = "w" if colour == "b" else "b"
224             piece.add_move_made()
225             if self.__board[end_position] != None:
```

```
                taken_piece = self.__board[end_position]
                special_case = "t"
                self.__pieces[op_colour].remove(end_position)
            self.update_board(piece,piece_position,end_position)

            if piece.return_name() == "King":
                self.update_king_position(colour,end_position)
                if self.__perspective == "w":
                    if abs(piece_position - end_position) == 2:
                        if end_position > piece_position:
                            self.update_board(self.__board[end_position+1],end_position+1,end_position-1)
                            special_case = "k"
                        else:
                            self.update_board(self.__board[end_position-2],end_position-2,end_position+1)
                            special_case = "q"

                if self.__perspective == "b":
                    if abs(piece_position - end_position) == 2:
                        if end_position < piece_position:
                            self.update_board(self.__board[end_position-1],end_position-1,end_position+1)
                            special_case = "k"
                        else:
                            self.update_board(self.__board[end_position+2],end_position+2,end_position-1)
                            special_case = "q"

            elif piece.return_name() == "Pawn":
                if end_position == self.__en_passant.peek():
                    if colour != self.__perspective:
                        t_square = end_position + N
                    else:
                        t_square = end_position-N
                    taken_piece = self.__board[t_square]
                    self.__board[t_square] = None
                    self.__pieces[op_colour].remove(t_square)
                    special_case = "e"
                elif abs(piece_position - end_position) == S+S:
                    if colour == self.__perspective:
                        en_passant = end_position - N
                    else:
                        en_passant = end_position + N
                elif end_position // 8 == 0 or end_position // 8 == 7:
                    promotion = "Q"
                    if promotion in self.__piece_classes:
                        new = self.__piece_classes[promotion]
                        rect = piece.return_rect()
                        self.__board[end_position] = new(colour,rect,self.__square_size)
                    temp = special_case
                    special_case = temp +"p"+ promotion
            self.__en_passant.push(en_passant)
            self.__move_log.push([piece_position,end_position,special_case,taken_piece])

        def handling_castling(self,start_square,end_square,colour):
            c_castling_rights = []
            if colour == "w":
                castle = [c for c in self.__castling_rights if c in ("K","Q")]
                if self.__perspective == "w":
                    k = 60
                    offset = 1
                else:
                    k = 3
                    offset = -1
            elif colour == "b":
                castle = [c for c in self.__castling_rights if c in ("k","q")]
                if self.__perspective == "w":
                    k = 4
                    offset = 1
                else:
                    k = 59
                    offset = -1
            if start_square != k or len(castle) == 0:
                return c_castling_rights
            for i in castle:
                if i.upper() == "K" and end_square == k+2*offset:
                    if self.__board[k+2*offset] == None and self.__board[k+3*offset] != None:
                        if self.__board[k+3*offset].return_algebraic() == "R" and
self.__board[k+3*offset].return_number_of_moves() == 0: #check if rook has moved
                            c_castling_rights.append((k,k+2*offset))
```

```
302                 if i.upper() == "Q" and end_square == k-2*offset:
303                     if self.__board[k-2*offset] == None and self.__board[k-3*offset] == None and self.__board[k-
4*offset] != None:
304                         if self.__board[k-4*offset].return_algebraic() == "R" and self.__board[k-
4*offset].return_number_of_moves() == 0:
305                             c_castling_rights.append((k,k-2*offset))
306             return c_castling_rights
307 
308 
309 
310         def all_possible_moves(self,colour):
311             if self.__perspective == colour:
312                 ub,lb = 7,0
313                 m = 1
314             else:
315                 ub,lb = 0,7
316                 m = -1
317 
318             for start_position in self.__pieces[colour]:
319                 piece = self.__board[start_position]
320                 algebraic = piece.return_algebraic()
321                 moves = self.__pieces_moves[algebraic]
322                 moves_types = self.__pieces_moves_types[algebraic]
323                 for move,move_type in zip(moves, moves_types):# iterates through the pieces moves - uses move_type
to check the the move contains certain direction
324                     end_position = start_position +m* move
325                     if 0 <= end_position <= 63:
326                         target_piece = self.__board[end_position]
327                         if algebraic == "p":
328                             row = start_position // 8
329                             if move == N and target_piece == None:
330                                 yield (start_position, end_position)
331 
332                                 end_position += m*move
333                                 if ((row == 6 and colour == self.__perspective) or (row == 1 and colour !=
self.__perspective)) and self.__board[end_position] == None:
334                                     yield (start_position, end_position)
335 
336                             elif target_piece != None and target_piece.return_colour() != colour:
337                                 column = start_position % 8
338                                 if E in move_type and ((column < 7 and m== 1) or (0 < column and m == -1)):
339                                     yield (start_position, end_position)
340 
341                                 elif W in move_type and ((0 < column and m == 1) or (column < 7 and m == -1)):
342                                     yield (start_position, end_position)
343 
344                             elif end_position == self.__en_passant.peek():
345                                 column = start_position % 8
346                                 if E in move_type and (( column < 7 and m == 1) or (0 < column and m == -1)):
347                                     yield (start_position, end_position)
348 
349                                 elif W in move_type and ((0 < column and m == 1) or (column < 7 and m == -1)):
350                                     yield (start_position, end_position)
351                             continue
352                         check = start_position
353                         while True: # check legal moves for sliding pieces
354                             if (check //8 == lb and N in move_type) or (check //8 == ub and S in move_type) or
(check %8 == lb and W in move_type) or (check %8 == ub and E in move_type):
355                                 break
356                             if not(0 <= end_position <= 63):
357                                 break
358                             target_piece = self.__board[end_position]
359                             if algebraic == "N":
360                                 if (check //8 == lb+m and move_type.count(N) == 2) or (check %8 == ub-m and
move_type.count(E) == 2) or (check %8 == lb +m and move_type.count(W) == 2) or (check //8 == ub -m and
move_type.count(S) == 2):
361                                     break
362                             elif target_piece == None:
363                                 yield (start_position, end_position)
364                             elif target_piece.return_colour() != colour:
365                                 yield (start_position, end_position)
366                             break
367 
368                         if target_piece == None:
369                             yield (start_position, end_position)
370                             if algebraic == "K":
371                                 break
```

```
                                check = end_position
                                end_position += m*move
                            else:
                                if target_piece.return_colour() != colour:
                                    yield (start_position, end_position)
                                break

        def all_legal_moves(self, colour): # converts pseudo-legal moves to legal moves
            self.__legal_moves = []
            all_moves = set(self.all_possible_moves(colour))
            op_colour = "w" if colour == "b" else "b"
            for move in all_moves:
                self.move(move)
                king_position = self.return_king_position(colour)
                all_op_moves =set(t[1] for t in set(self.all_possible_moves(op_colour)))
                if king_position not in all_op_moves: # sees if the opponent can take the king in the next move if
they can then it is not a legal move
                    self.__legal_moves.append(move)
                    if king_position == move[1] and abs(move[0] - move[1]) == 1 and self.__move_log.peek()[2] ==
"":
                        if move[0] not in all_op_moves and (2 * move[1] - move[0]) not in all_op_moves and
self.__board[king_position].return_number_of_moves() == 1:
                            if self.handling_castling(move[0], 2 * move[1] - move[0], colour):
                                self.__legal_moves.append((move[0], 2 * move[1] - move[0]))
                self.undo_move()
            return self.__legal_moves

        def is_check(self,colour):
            if colour == "w":
                op_colour = "b"
                king_pos = self.__position_white_king
            else:
                op_colour = "w"
                king_pos = self.__position_black_king
            op_moves = set(self.all_possible_moves(op_colour))
            op_moves_end = set(t[1] for t in op_moves)
            if king_pos in op_moves_end:
                return True , op_moves
            else:
                return False , op_moves

        def is_checkmate(self,colour,op_moves):
            if colour == "w":
                king_pos = self.__position_white_king
            else:
                king_pos = self.__position_black_king
            op_moves_end = set(t[1] for t in op_moves)
            if king_pos in op_moves_end:
                return True
            else:
                return False

        def return_all_colour_pieces_positions(self,colour):
            return self.__pieces[colour]

        def return_piece(self,position):
            return self.__board[position]

        def displaying_possible_moves(self, screen, piece_position):
            p_moves = [t for t in self.__legal_moves if t[0] == piece_position]
            if len(p_moves) > 0:
                for i in range(len(p_moves)):
                    square = p_moves[i][1]
                    row = square // 8
                    column = square % 8
                    rect = (column * self.__square_size, row * self.__square_size, self.__square_size,
self.__square_size)
                    pygame.draw.rect(screen, self.__boardcolour3, rect)

        def update_king_position(self,colour,position):
            if colour == "b":
                self.__position_black_king = position
            else:
                self.__position_white_king = position

        def return_king_position(self,colour):
            if colour == "b":
```

```
                return self.__position_black_king
            elif colour == "w":
                return self.__position_white_king

        def undo_move(self): # undoes the move / maintains castling rights, en passant etc
            if self.__move_log.length() == 0:
                return
            move = self.__move_log.pop()
            piece = self.__board[move[1]]
            piece.remove_move_made()
            colour = piece.return_colour()
            t_piece = move[3]
            self.update_board(piece,move[1],move[0])
            if piece.return_algebraic() == "K":
                self.update_king_position(colour,move[0])
            if move[2] != "":
                if move[2] in "t":
                    self.update_board(move[3],move[1],move[1])
                elif move[2] == "e":

                    if colour != self.__perspective:
                        t_square = move[1] + N
                    else:
                        t_square =move[1]-N
                    self.update_board(move[3],t_square,t_square)
                elif "p" in move[2] :
                    self.__pieces[colour].remove(move[0])
                    self.update_board(Pawn(colour,piece.return_rect(),self.__square_size),move[0],move[0])

                    if t_piece != None:
                        self.update_board(t_piece,move[1],move[1])
                elif move[2] in "kq":
                    if self.__perspective == "w":
                        if move[1] > move[0]:
                            self.update_board(self.__board[move[1]-1],move[1]-1,move[1]+1)
                        else:
                            self.update_board(self.__board[move[1]+1],move[1]+1,move[1]-2)
                    if self.__perspective == "b":
                        if move[1] < move[0]:
                            self.update_board(self.__board[move[1]+1],move[1]+1,move[1]-1)
                        else:
                            self.update_board(self.__board[move[1]-1],move[1]-1,move[1]+2)
            self.__en_passant.pop()

        def update_rects(self):
            for i in range(len(self.__board)):
                piece = self.__board[i]
                if piece != None:
                    self.update_piece_rect(piece,i)

        def validate_user_move(self,move):
            if move in self.__legal_moves:
                return True
            else:
                return False

        def convert_move_log_to_notation(self): # Algorithms used to convert move made into chess notation
            move = self.__move_log.peek()
            prev_file = self.__columns_to_files[move[0]%8]
            notation_x = self.__columns_to_files[move[1]%8]
            notation_y = self.__rows_to_rank[move[1]//8]
            piece = self.return_piece(move[1])
            notation = ""
            if move[2] != "":
                if move[2][0] == "t" or move[2] == "e": # t = taken, e = en passant, q = queen side castling, k =
king side castling
                    if piece.return_algebraic() == "p" or "p" in move[2]:
                        notation = (prev_file + "x"+ notation_x + notation_y)
                    else:
                        notation = (piece.return_algebraic() + "x" + notation_x + notation_y)
                if move[2] == "q":
                    notation = ("O-O-O")
                if move[2] == "k":
                    notation = ("O-O")
                if "p" in move[2]:
                    if len(notation) == 0:
                        notation = notation_x + notation_y
```

```
                    temp = notation
                    notation = temp + move[2][move[2].index("p") +1 ]
            else:
                if piece.return_algebraic() == "p":
                    notation = (notation_x + notation_y)
                else:
                    notation = (piece.return_algebraic() + notation_x + notation_y)
            if self.__move_log.length()%2 == 0:
                opposite_colour = "w"
            else:
                opposite_colour = "b"
            check, moves = self.is_check(opposite_colour)
            if check:
                notation = notation + "+"
            self.all_legal_moves(opposite_colour)
            if len(self.__legal_moves) == 0:
                if self.is_checkmate(opposite_colour,moves):
                    notation = notation + "+"
            self.__move_record.push(notation)

        def check_if_stalemate(self):
            if len(self.__legal_moves) == 0:
                return True

        def check_if_draw_by_repetition(self): # checks if the same position has been repeated 3 times
            if self.__move_log.length() > 6:
                last_six_move_logs = []
                for i in range(6):
                    last_six_move_logs.append(self.__move_log.pop())
                last_six_moves = [(t[0], t[1]) for t in last_six_move_logs]
                reverse_direction = [(b,a) for a,b in last_six_moves[2:4]]
                repeated = last_six_moves[0:2] == reverse_direction == last_six_moves[4:6]
                for move in reversed(last_six_move_logs):
                    self.__move_log.push(move)
                return repeated
            return False

        def check_if_end_game(self): # if end game change evaluation method
            self.__end_game = False
            count = 0
            for pos in self.__pieces["w"]:
                if self.__board[pos].return_algebraic() not in ["K","p"]:
                    count += 1
            if count <=3:
                self.__board[self.__position_white_king].set_piece_square_table_end_game()
                self.__end_game = True
            else:
                self.__board[self.__position_white_king].set_piece_square_table()
            count = 0
            for pos in self.__pieces["b"]:
                if  self.__board[pos].return_algebraic() not in ["K","p"]:
                    count += 1
            if count <=3:
                self.__board[self.__position_black_king].set_piece_square_table_end_game()
                self.__end_game = True
            else:
                self.__board[self.__position_black_king].set_piece_square_table()

        def convert_puzzle_move_to_move(self,move):
            for key, value in self.__columns_to_files.items():
                if move[0] == value:
                    column = key
                    break
            for key, value in self.__rows_to_rank.items():
                if move[1] == value:
                    row = key
                    break
            return (row*8 + column)

        def is_end_game(self):
            return self.__end_game

        def return_move_record(self):
            return self.__move_record

        def return_move_log(self):
            return self.__move_log
```

```
598
599         def return_moves(self):
600             return self.__legal_moves
601
602         def undo_move_record(self):
603             if self.__move_record.length() > 0:
604                 self.__move_record.pop()
```

engine.py

```
001     from time import time
002
003     class Engine():
004         def __init__(self,board,AI_colour,max_depth):
005             self.__board = board
006             self.__AI_colour = AI_colour
007             self.__max_depth = max_depth
008             self.__current_best_move = None
009
010         def update_board(self,board):
011             self.__board = board
012         def Minimax(self,depth,maximising_player,alpha,beta): # + using alpha beta pruning
013             moves = []
014             if depth == 0: # check if max depth
015                 return None,self.evaluate_position()
016             colour = "w" if maximising_player else "b"
017             self.__board.all_legal_moves(colour)
018             moves = self.__board.return_moves()
019             if len(moves) ==0: # check if game is over
020                 op_colour = "w" if colour == "b" else "b"
021                 m = list(self.__board.all_possible_moves(op_colour))
022                 if self.__board.is_checkmate(colour,m):
023                     evaluation = -10000 if colour == "w" else 10000
024                 else:
025                     evaluation = 0
026                 return None,evaluation
027             if self.__board.check_if_draw_by_repetition():
028                 return None,0
029             moves = self.order_moves(moves)
030             if self.__search_depth == self.__max_depth and self.__search_depth == depth: # ensures that the
previous depth search best move is searched first
031                 if self.__current_best_move in moves:
032                     moves.remove(self.__current_best_move)
033                     moves.insert(0, self.__current_best_move)
034             if maximising_player: # maximising player wants a more position evaluation
035                 max_evaluation = -100000
036                 for move in moves:
037                     self.__board.move(move)
038                     evaluation = self.Minimax(depth-1,False,alpha,beta) # recursively calls the Minimax function –
at the higher depth
039                     self.__board.undo_move()
040                     if evaluation[1] > max_evaluation:
041                         max_evaluation = evaluation[1]
042                         best_move = move
043                     alpha = max(alpha,max_evaluation)
044                     if beta <= alpha: #pruning worse branches
045                         break
046                 return best_move,max_evaluation
047             else: # maximising player wants a more negative evaluation
048                 min_evaluation = 100000
049                 for move in moves:
050                     self.__board.move(move)
051                     evaluation = self.Minimax(depth-1,True,alpha,beta)
052                     self.__board.undo_move()
053                     if evaluation[1] < min_evaluation:
054                         min_evaluation = evaluation[1]
055                         best_move = move
056                     beta = min(beta,min_evaluation)
057                     if beta <= alpha:
058                         break
059                 return best_move,min_evaluation
060
061         def iterative_deepening(self,maximising_player): # Optimisation - Iteratively calls Minimax - increasing
the depth https://www.chessprogramming.org/Iterative_Deepening
062             self.__current_best_move = None
063             start_time = time()
064             self.__search_depth = 0
065             while True:
066                 self.__current_best_move, self.__current_best_eval = self.Minimax(self.__search_depth,
maximising_player, -1000000, 1000000)
067                 if self.__search_depth == self.__max_depth or time() - start_time > 10: # if time to make move
exceeds limits/max depth reached then stop
068                     break
069                 self.__search_depth += 1
070             return self.__current_best_move, self.__current_best_eval
071
```

```
072         def order_moves(self, moves): # shallow evaluation to order the moves
073             move_predictions = sorted(
074                 ((move, (self.__board.return_piece(move[1]).return_value()-
self.__board.return_piece(move[0]).return_value())  if self.__board.return_piece(move[1]) is not None else 0) for move
in moves),
075                 key=lambda x: x[1],
076                 reverse=True
077             ) # orders move by the difference between the taken piece and piece taking it
078             ordered_moves = [move for move, prediction in move_predictions]
079             return ordered_moves
080 
081         def evaluate_position(self):
082             evaluation = 0
083             pos = self.__board.return_all_colour_pieces_positions("w")
084             for i in pos:
085                 piece= self.__board.return_piece(i)
086                 evaluation += piece.return_value()
087                 evaluation += piece.return_piece_square_table(i)
088             pos = self.__board.return_all_colour_pieces_positions("b")
089             for i in pos:
090                 piece= self.__board.return_piece(i)
091                 evaluation -= piece.return_value()
092                 evaluation -= piece.return_piece_square_table(63-i)
093             if self.__board.is_end_game():
094                 evaluation += self.end_game_evaluation()
095             return evaluation
096 
097         def end_game_evaluation(self): # Encourages the king to move towards each other
098             w_king_position = self.__board.return_king_position("w")
099             b_king_position = self.__board.return_king_position("b")
100             king_distance = self.manhattan_distance(w_king_position, b_king_position)
101             if self.__AI_colour == "w":
102                 return 4 * (14- king_distance) # tested with 4 and it works the best
103             else:
104                 return -4 * (14 - king_distance)
105 
106         def manhattan_distance(self,position1,position2): # Calculates the distance between two positions on the
board
107             x1,y1 = position1//8,position1%8
108             x2,y2 = position2//8,position2%8
109             return abs(x1-x2) + abs(y1-y2)
```

piece.py

```
001     import pygame
002
003     class Piece():
004         def __init__(self,name,colour,value,algebraic,rect,square_size):
005             self.__square_size = square_size
006             self.__name = name
007             self.__colour = colour
008             self.__value = value
009             self.__algebraic = algebraic
010             self.__rect = rect
011             self.__num_moves = 0
012
013         def return_value(self):
014             return self.__value
015         def display_piece(self,screen):
016             surf = pygame.transform.smoothscale(pygame.transform.scale(pygame.image.load("Pieces/"+ self.__colour
+ self.__algebraic +".png"),(self.__square_size,self.__square_size)).convert_alpha(),
(self.__square_size,self.__square_size))
017             screen.blit(surf,self.__rect)
018         def return_colour(self):
019             return self.__colour
020         def drag_rect(self,relative_position):
021             rect_x = self.__rect[0] + relative_position[0]
022             rect_y = self.__rect[1] + relative_position[1]
023             self.__rect = (rect_x,rect_y)
024         def update_rect(self,rect):
025             self.__rect = rect
026         def return_algebraic(self):
027             return self.__algebraic
028         def return_name(self):
029             return self.__name
030         def add_move_made(self):
031             self.__num_moves +=1
032         def remove_move_made(self):
033             self.__num_moves -=1
034         def return_number_of_moves(self):
035             return self.__num_moves
036         def return_rect(self):
037             return self.__rect
038     class Rook(Piece):
039         def __init__ (self, colour,rect,square_size):
040             super().__init__("Rook",colour,500,"R",rect,square_size)
041             self.__evaluation_squares = [0,  0,  0,  0,  0,  0,  0,  0,
042                                          5, 10, 10, 10, 10, 10, 10,  5,
043                                         -5,  0,  0,  0,  0,  0,  0, -5,
044                                         -5,  0,  0,  0,  0,  0,  0, -5,
045                                         -5,  0,  0,  0,  0,  0,  0, -5,
046                                         -5,  0,  0,  0,  0,  0,  0, -5,
047                                         -5,  0,  0,  0,  0,  0,  0, -5,
048                                          0,  0,  0,  5,  5,  0,  0,  0]
049         def rotate_piece_square_table(self):
050             self.__evaluation_squares.reverse()
051         def return_piece_square_table(self,position):
052             return self.__evaluation_squares[position]
053     class Bishop(Piece):
054         def __init__ (self, colour, rect,square_size):
055             super().__init__("Bishop",colour,330,"B",rect,square_size)
056             self.__evaluation_squares = [-20,-10,-10,-10,-10,-10,-10,-20,
057                                         -10,  0,  0,  0,  0,  0,  0,-10,
058                                         -10,  0,  5, 10, 10,  5,  0,-10,
059                                         -10,  5,  5, 10, 10,  5,  5,-10,
060                                         -10,  0, 10, 10, 10, 10,  0,-10,
061                                         -10, 10, 10, 10, 10, 10, 10,-10,
062                                         -10,  5,  0,  0,  0,  0,  5,-10,
063                                         -20,-10,-10,-10,-10,-10,-10,-20]
064         def return_piece_square_table(self,position):
065             return self.__evaluation_squares[position]
066         def rotate_piece_square_table(self):
067             self.__evaluation_squares.reverse()
068     class Knight(Piece):
069         def __init__ (self, colour,rect,square_size):
070             super().__init__("Knight",colour,320,"N",rect,square_size)
071             self.__evaluation_squares = [-50,-40,-30,-30,-30,-30,-40,-50,
072                                         -40,-20,  0,  0,  0,  0,-20,-40,
073                                         -30,  0, 10, 15, 15, 10,  0,-30,
074                                         -30,  5, 15, 20, 20, 15,  5,-30,
```

```
                                    -30,  0, 15, 20, 20, 15,  0,-30,
                                    -30,  5, 10, 15, 15, 10,  5,-30,
                                    -40,-20,  0,  5,  5,  0,-20,-40,
                                    -50,-40,-30,-30,-30,-30,-40,-50]

        def return_piece_square_table(self,position):
            return self.__evaluation_squares[position]
        def rotate_piece_square_table(self):
            self.__evaluation_squares.reverse()
    class Queen(Piece):
        def __init__ (self, colour,rect,square_size):
            super().__init__("Queen",colour,900,"Q",rect,square_size)
            self.__evaluation_squares = [-20,-10,-10, -5, -5,-10,-10,-20,
                                    -10,  0,  0,  0,  0,  0,  0,-10,
                                    -10,  0,  5,  5,  5,  5,  0,-10,
                                    -5,  0,  5,  5,  5,  5,  0, -5,
                                     0,  0,  5,  5,  5,  5,  0, -5,
                                    -10,  5,  5,  5,  5,  5,  0,-10,
                                    -10,  0,  5,  0,  0,  0,  0,-10,
                                    -20,-10,-10, -5, -5,-10,-10,-20]
        def return_piece_square_table(self,position):
            return self.__evaluation_squares[position]
        def rotate_piece_square_table(self):
            self.__evaluation_squares.reverse()
    class King(Piece):
        def __init__ (self, colour,rect,square_size):
            super().__init__("King",colour,20000,"K",rect,square_size)
            self.__evaluation_squares= [-30,-40,-40,-50,-50,-40,-40,-30,
                                    -30,-40,-40,-50,-50,-40,-40,-30,
                                    -30,-40,-40,-50,-50,-40,-40,-30,
                                    -30,-40,-40,-50,-50,-40,-40,-30,
                                    -20,-30,-30,-40,-40,-30,-30,-20,
                                    -10,-20,-20,-20,-20,-20,-20,-10,
                                    20, 20,  0,  0,  0,  0, 20, 20,
                                    20, 30, 10,  0,  0, 10, 30, 20 ]
            self.__evaluation_squares_end_game = [-50,-40,-30,-20,-20,-30,-40,-50,
                                                  -30,-20,-10,  0,  0,-10,-20,-30,
                                                  -30,-10, 20, 30, 30, 20,-10,-30,
                                                  -30,-10, 30, 40, 40, 30,-10,-30,
                                                  -30,-10, 30, 40, 40, 30,-10,-30,
                                                  -30,-10, 20, 30, 30, 20,-10,-30,
                                                  -30,-30,  0,  0,  0,  0,-30,-30,
                                                  -50,-30,-30,-30,-30,-30,-30,-50]
            self.__current_evaluation_squares = self.__evaluation_squares
        def return_piece_square_table(self,position):
            return self.__current_evaluation_squares[position]
        def set_piece_square_table_end_game(self):
            self.__current_evaluation_squares = self.__evaluation_squares_end_game
        def is_in_end_game(self):
            if self.__current_evaluation_squares == self.__evaluation_squares_end_game:
                return True
            return False
        def set_piece_square_table(self):
            self.__current_evaluation_squares = self.__evaluation_squares
        def rotate_piece_square_table(self):
            self.__evaluation_squares.reverse()

    class Pawn(Piece):
        def __init__ (self, colour,rect,square_size):
            super().__init__("Pawn",colour,100,"p",rect,square_size)
            self.__evaluation_squares = [0,  0,  0,  0,  0,  0,  0,  0,
                                    50, 50, 50, 50, 50, 50, 50, 50,
                                    10, 10, 20, 30, 30, 20, 10, 10,
                                    5,  5, 10, 25, 25, 10,  5,  5,
                                    0,  0,  0, 20, 20,  0,  0,  0,
                                    5, -5,-10,  0,  0,-10, -5,  5,
                                    5, 10, 10,-20,-20, 10, 10,  5,
                                    0,  0,  0,  0,  0,  0,  0,  0]
        def return_piece_square_table(self,position):
            return self.__evaluation_squares[position]
        def rotate_piece_square_table(self):
            self.__evaluation_squares.reverse()
```

mouse.py

```
01      import pygame
02
03      class Mouse():
04          def __init__(self):
05              self.__initial_position_x = 0
06              self.__initial_position_y = 0
07              self.__final_position_x = 0
08              self.__final_position_y = 0
09              self.__selected_piece = None
10          def update_initial_position(self):
11              self.__initial_position_x,self.__initial_position_y = pygame.mouse.get_pos()
12          def return_initial_position(self):
13              return self.__initial_position_x,self.__initial_position_y
14          def update_final_position(self):
15              self.__final_position_x,self.__final_position_y = pygame.mouse.get_pos()
16          def return_final_position(self):
17              return self.__final_position_x,self.__final_position_y
18          def update_selected_piece(self,piece):
19              self.__selected_piece = piece
20          def return_selected_piece(self):
21              return self.__selected_piece
22          def drag(self,rel_position):
23              if self.__selected_piece != None:
24                  self.__selected_piece.drag_rect(rel_position)
25          def reset(self):
26              self.__initial_position_x = 0
27              self.__initial_position_y = 0
28              self.__final_position_x = 0
29              self.__final_position_y = 0
30              self.__selected_piece = None
```

stack.py

```
01      class Node:
02          def __init__(self, data=None): # each node stores the data and the pointer to the next item in the stack
03              self.__data = data
04              self.__next = None
05
06          def get_data(self):
07              return self.__data
08
09          def set_data(self, data):
10              self.__data = data
11
12          def get_next(self):
13              return self.__next
14
15          def set_next(self,next):
16              self.__next = next
17
18
19      class Stack:
20          def __init__(self):
21              self.__top = None
22
23          def push(self, data): # add a new node to the top of the stack
24              if self.__top is None:
25                  self.__top = Node(data)
26              else:
27                  new_node = Node(data)
28                  new_node.set_next(self.__top)
29                  self.__top = new_node
30
31          def pop(self): # remove the top node from the stack and return its data
32              if self.__top is None:
33                  return None
34              else:
35                  popped_node = self.__top
36                  self.__top = self.__top.get_next()
37                  popped_node.set_next(None)
38                  return popped_node.get_data()
39
40          def peek(self): # return the data of the top node in the stack
41              return self.__top.get_data() if self.__top is not None else None
42
43          def length(self): # return the number of nodes in the stack
44              current = self.__top
45              count = 0
46              while current:
47                  count += 1
48                  current = current.get_next()
49              return count
50
51          def is_empty(self): # return True if the stack is empty, False otherwise
52              return self.__top is None
```

chessclient.py

```
01      import http.client
02      import json
03      from urllib.parse import quote
04      import http.server
05
06      class ChessClient:
07          @staticmethod
08          def create_user(username, salt, hashed_password):
09              conn = http.client.HTTPSConnection("17mlee.eu.pythonanywhere.com")
10              data = {"username": username, "salt": salt, "hashed_password": hashed_password}
11              headers = {'Content-Type': 'application/json'} # add a header to the http request – data in json
12              conn.request("POST", "/users", body=json.dumps(data), headers=headers)
13              response = conn.getresponse()
14              data = response.read().decode()
15              return json.loads(data) # read file using json
16          @staticmethod
17          def get_salt(username):
18              conn = http.client.HTTPSConnection("17mlee.eu.pythonanywhere.com")
19              params = "?username=" + quote(username)
20              conn.request("GET", "/users/salt" + params)
21              response = conn.getresponse()
22              data = response.read().decode()
23              return json.loads(data)
24          @staticmethod
25          def validate_user(username, hashed_password):
26              conn = http.client.HTTPSConnection("17mlee.eu.pythonanywhere.com")
27              params = "?username=" + quote(username) + "&hashed_password=" + quote(hashed_password)
28              conn.request("GET", "/users/validate" + params) # setting up the parameters for the search
29              response = conn.getresponse()
30              data = response.read().decode()
31              return json.loads(data)
32          @staticmethod
33          def get_puzzle (userID, rating_min, rating_max):
34              conn = http.client.HTTPSConnection("17mlee.eu.pythonanywhere.com")
35              params = "?userID=" + str(userID) + "&rating_max=" + str(rating_max) + "&rating_min=" +
str(rating_min)
36              conn.request("GET", "/puzzles" + params)
37              response = conn.getresponse()
38              data = response.read().decode()
39              return json.loads(data)
40          @staticmethod
41          def puzzle_completed(userID, puzzleID):
42              conn = http.client.HTTPSConnection("17mlee.eu.pythonanywhere.com")
43              data = {"userID": userID, "puzzleID": puzzleID}
44              headers = {'Content-Type': 'application/json'}
45              conn.request("POST", "/puzzles", body=json.dumps(data), headers=headers)
46              response = conn.getresponse()
47              data = response.read().decode()
48          @staticmethod
49          def get_games_name(userID,result,playedas,AI):
50              conn = http.client.HTTPSConnection("17mlee.eu.pythonanywhere.com")
51              params = "?userID=" + str(userID)
52              for i in range(len(result)):
53                  params += "&result=" + result[i]
54              for i in range(len(playedas)):
55                  params += "&playedas=" + playedas[i]
56              for i in range(len(AI)):
57                  params += "&AI=" + str(AI[i])
58              conn.request("GET", "/games/names" + params)
59              response = conn.getresponse()
60              data = response.read().decode()
61              return json.loads(data)
62          @staticmethod
63          def get_game_moves(gameID):
64              conn = http.client.HTTPSConnection("17mlee.eu.pythonanywhere.com")
65              params = "?gameID=" + str(gameID)
66              conn.request("GET", "/games" + params)
67              response = conn.getresponse()
68              data = response.read().decode()
69              return json.loads(data)
70          @staticmethod
71          def save_game(userID,name, moves,result,playedas,AI):
72              conn = http.client.HTTPSConnection("17mlee.eu.pythonanywhere.com")
73              data = {"userID": userID, "name": name, "moves": moves, "result": result, "playedas": playedas, "AI":
AI}
74              headers = {'Content-Type': 'application/json'}
```

```
75              conn.request("POST", "/games", body=json.dumps(data), headers=headers)
76              response = conn.getresponse()
77              data = response.read().decode()
```

Testing

https://colytongrammarschool.sharepoint.com/sites/ComputingNEA/SitePages/MLee.aspx

Test No.	Objective (if applicable)	Description of test	Type of test	Test Data	Expected result	Actual result	Video time stamp
Test Video 1							
1	1bi	Attempt to create account where the password contains less than 8 characters.	Erroneous	Username: "TestUser" Password: "Qwerty1"	Error message: "Password needs to be at least 8 characters long"	Error message: "Password needs to be at least 8 characters long"	0:02
2	1bii	Attempt to create account where the password doesn't contain a special character.	Erroneous	Username: "TestUser" Password: "Qwerty12"	Error message: "Password needs to contain at least 1 special character"	Error message: "Password needs to contain at least 1 special character"	0:04
3	1	Attempt to create a valid account.	Boundary	Username: "TestUser" Password: "Qwerty@1"	Message: "Inputted user details are valid"	Message: "Inputted user details are valid"	0:05
4	1a	Attempt to create account where the username is not unique.	Erroneous	Username: "TestUser" Password: "Qwerty@1"	Error message: "Username already exists"	Error message: "Username already exists"	0:09
5	2a	Attempt to log in to an account which does not exist.	Erroneous	Username: "TestUser1" Password: "Pa$$word"	Error message: "Username does not exist"	Error message: "Username does not exist"	0:11
6	2b	Attempt to log in to the account where the password is incorrect.	Erroneous	Username: "TestUser" Password: "Qwerty"	Error message: "Incorrect Password"	Error message: "Incorrect Password"	0:16
7	2	Attempt to log in to the account where the details are correct.	Typical	Username: "TestUser" Password: "Qwerty@1"	User is authenticated. User is then directed to the Main Menu screen.	User is authenticated. User is then directed to the Main Menu screen.	0:20
8	3aii,5a,5b	Attempt to play a local game.	Typical	Click the Button "Play Against Human"	User is directed to the Game Menu. The chess pieces are	User is directed to the Game Menu. The chess pieces are	0:23

					set up in the correct positions on the board.	set up in the correct positions on the board.	
9	4a	Attempt to make the black pieces move first.	Erroneous	Try to drag and drop a black piece.	The user is not able to pick up the piece.	The user is not able to pick up the piece.	0:32
10	4a	Attempt to make the white pieces move first.	Typical	Try to drag and drop a white piece.	The user can pick up the piece.	The user can pick up the piece and drag the piece.	0:35
12	4b,5c,5e,5f	Attempt to make alternating moves of black and white.	Typical	Try to drag and drop a black piece after the white pieces have moved.	The user can pick up the piece.	The user can pick up the piece.	0:36
13	4ci	Attempt to move the pawn by two squares when it is the pawn's first move, and the two squares are empty.	Typical	For the first move of the pawn, try and drag and drop the pawn 2 squares forwards when the two squares are empty.	The game highlights the possible moves. The two squares ahead of the pawn are highlighted. When the user drops the piece in the square, the piece stays in the square. The move log is updated to show the move made in chess notation.	The game highlights the possible moves. The two squares ahead of the pawn are highlighted. When the user drops the piece in the square, the piece stays in the square. The move log is updated to show the move made in chess notation.	0:34
14	4ci	Attempt to move the pawn by two squares forward when it is the pawn's first move, and the one of the two squares are not empty.	Erroneous	For the first move of the pawn, try and drag and drop the pawn 2 squares forwards when the two squares are not empty.	The game highlights the moves that the pawn can make, if any. When the user drops the piece in the square, the piece does not remain in the square and returns to the original square. The player can make another move.	The game highlights the moves that the pawn can make, if any. When the user drops the piece in the square, the piece does not remain in the square and returns to the original square. The player can make another move.	0:49
15	4cii	Attempt to move the pawn by one square forward, and the square is occupied.	Erroneous	Try and drag and drop the pawn 1 square forwards when the square ahead is empty.	The game does not highlight the moves ahead of the pawn. When the user drops the piece in the square, the piece does not remain in the square and returns to the original square.	The game does not highlight the moves ahead of the pawn. When the user drops the piece in the square, the piece does not remain in the square and returns to the original square.	0:53

					The player can make another move.	The player can make another move.	
16	4cii	Attempt to move the pawn by one square forward, and the square are empty.	Typical	Try and drag and drop the pawn 1 square forwards when the square ahead is empty.	The game highlights the moves that the pawn can make. The square ahead of the pawn is highlighted. When the user drops the piece in the square, the piece stays in the square. The move log is updated to show the move made in chess notation.	The game highlights the moves that the pawn can make. The square ahead of the pawn is highlighted. When the user drops the piece in the square, the piece stays in the square. The move log is updated to show the move made in chess notation.	0:50
17	4ciii	Attempt to move the pawn by diagonally by one square, and the square is empty.	Erroneous	Try and drag and drop the pawn 1 square diagonally forwards when the square is empty.	The game does not highlight the diagonal square. When the user drops the piece in the square, the piece does not remain in the square and returns to the original square. The player can make another move.	The game does not highlight the diagonal square. When the user drops the piece in the square, the piece does not remain in the square and returns to the original square. The player can make another move.	0:56
18	4ciii	Attempt to move the pawn diagonally by one square, and the square contains an opposition piece.	Typical	Try and drag and drop the pawn 1 square diagonally forwards when the square contains an opposition piece.	The game highlights the possible squares. When the user drops the piece in the square, the piece stays in the square and the piece originally there disappears. The move log is updated to show the move made in chess notation.	The game highlights the possible squares. When the user drops the piece in the square, the piece stays in the square and the piece originally there disappears. The move log is updated to show the move made in chess notation.	0:57
19	4civ	Attempt to move a knight in an L shape when the square is empty.	Typical	Try and drag and drop the knight 2 squares horizontally or vertically and 1 square in a perpendicular direction	The game highlights the square. When the user drops the piece in the square, the piece stays in the square. The move log is updated to show	The game highlights the square. When the user drops the piece in the square, the piece stays in the square. The move log is updated to show	1:03

				when the end square is empty.	the move made in chess notation.	the move made in chess notation.	
20	4civ	Attempt to move a knight in an L shape when the square contains a same colour piece.	Erroneous	Try and drag and drop the knight 2 squares horizontally or vertically and 1 square in a perpendicular direction when the end square contains a same colour piece.	The game does not highlight the square. When the user drops the piece in the square, the piece does not remain in the square and returns to the original square. The player can make another move.	The game does not highlight the square. When the user drops the piece in the square, the piece does not remain in the square and returns to the original square. The player can make another move.	1:10
21	4civ	Attempt to move a knight in an L shape when the square contains an opposition colour piece.	Typical	Try and drag and drop the knight 2 squares horizontally or vertically and 1 square in a perpendicular direction when the end square contains an opposition piece.	The game highlights the possible squares. When the user drops the piece in the square, the piece stays in the square and the piece originally there disappears. The move log is updated to show the move made in chess notation.	The game highlights the possible squares. When the user drops the piece in the square, the piece stays in the square and the piece originally there disappears. The move log is updated to show the move made in chess notation.	1:14
22	4cv	Attempt to move a bishop diagonally when the bishop is blocked by its own pieces.	Erroneous	Try and drag and drop the bishop to a diagonal square from the start position when the squares to the end square are not empty.	The game highlights the possible squares before the bishop is blocked by its own pieces. When the user drops the piece in the square, the piece does not remain in the square and returns to the original square. The player can make another move.	The game highlights the possible squares before the queen is blocked by its own pieces. When the user drops the piece in the square, the piece does not remain in the square and returns to the original square. The player can make another move.	1:22
23	4cv	Attempt to move a bishop diagonally when the bishop is not blocked by its own pieces.	Typical	Try and drag and drop the bishop to a diagonal square from the start position when the squares to the end square are empty.	The game highlights the possible squares. When the user drops the piece in the square, the piece stays in the square. The move log is	The game highlights the possible squares. When the user drops the piece in the square, the piece stays in the square. The move log is	1:30

					updated to show the move made in chess notation.	updated to show the move made in chess notation.	
24	4cv	Attempt to move a bishop so that it takes an opposition piece.	Typical	Try and drag and drop the bishop to a diagonal square from the start position when the squares to the end square are empty and the end square has an opposition piece.	The game highlights the possible squares. When the user drops the piece in the square, the piece stays in the square and the piece originally there disappears. The move log is updated to show the move made in chess notation.	The game highlights the possible squares. When the user drops the piece in the square, the piece stays in the square and the piece originally there disappears. The move log is updated to show the move made in chess notation.	1:43
25	4cvi	Attempt to move a rook when the rook is blocked by its own pieces.	Erroneous	Try and drag and drop the rook to a vertical or horizontal square from the start position when the squares to the end square are not empty.	The game highlights the possible squares before the rook is blocked by its own pieces. When the user drops the piece in the square, the piece does not remain in the square and returns to the original square. The player can make another move.	The game highlights the possible squares before the rook is blocked by its own pieces. When the user drops the piece in the square, the piece does not remain in the square and returns to the original square. The player can make another move.	1:57
26	4cvi	Attempt to move a rook vertically when the rook is not blocked by its own pieces.	Typical	Try and drag and drop the rook to a vertical square from the start position when the squares to the end square are empty.	The game highlights the possible squares. When the user drops the piece in the square, the piece stays in the square. The move log is updated to show the move made in chess notation.	The game highlights the possible squares. When the user drops the piece in the square, the piece stays in the square. The move log is updated to show the move made in chess notation.	2:03
27	4cvi	Attempt to move a rook horizontally when the rook is not blocked by its own pieces.	Typical	Try and drag and drop the rook to a horizontal square from the start position when the squares to the end square are empty.	The game highlights the possible squares. When the user drops the piece in the square, the piece stays in the square. The move log is updated to show the move made in chess notation.	The game highlights the possible squares. When the user drops the piece in the square, the piece stays in the square. The move log is updated to show the move made in chess notation.	2:09

28	4cvi	Attempt to move a rook so that it takes an opposition piece.	Typical	Try and drag and drop the rook to a square from the start position when the squares to the end square are empty and the end square has an opposition piece.	The game highlights the possible squares. When the user drops the piece in the square, the piece stays in the square and the piece originally there disappears. The move log is updated to show the move made in chess notation.	The game highlights the possible squares. When the user drops the piece in the square, the piece stays in the square and the piece originally there disappears. The move log is updated to show the move made in chess notation.	2:15
29	4cvii	Attempt to move a queen when the queen is blocked by its own pieces.	Erroneous	Try and drag and drop the queen to a vertical or horizontal or diagonal square from the start position when the squares to the end square are not empty.	The game highlights the possible squares before the queen is blocked by its own pieces. When the user drops the piece in the square, the piece does not remain in the square and returns to the original square. The player can make another move.	The game highlights the possible squares before the queen is blocked by its own pieces. When the user drops the piece in the square, the piece does not remain in the square and returns to the original square. The player can make another move.	2:27
30	4cvii	Attempt to move the queen vertically when the queen is not blocked by its own pieces.	Typical	Try and drag and drop the queen to a vertical square from the start position when the squares to the end square are empty.	The game highlights the possible squares. When the user drops the piece in the square, the piece stays in the square. The move log is updated to show the move made in chess notation.	The game highlights the possible squares. When the user drops the piece in the square, the piece stays in the square. The move log is updated to show the move made in chess notation.	2:30
31	4cvii	Attempt to move the queen horizontally when the queen is not blocked by its own pieces.	Typical	Try and drag and drop the queen to a horizontal square from the start position when the squares to the end square are empty.	The game highlights the possible squares. When the user drops the piece in the square, the piece stays in the square. The move log is updated to show the move made in chess notation.	The game highlights the possible squares. When the user drops the piece in the square, the piece stays in the square. The move log is updated to show the move made in chess notation.	2:36

32	4cvii	Attempt to move the queen diagonally when the queen is not blocked by its own pieces.	Typical	Try and drag and drop the queen to a diagonal square from the start position when the squares to the end square are empty.	The game highlights the possible squares. When the user drops the piece in the square, the piece stays in the square. The move log is updated to show the move made in chess notation.	The game highlights the possible squares. When the user drops the piece in the square, the piece stays in the square. The move log is updated to show the move made in chess notation.	2:34
33	4cvii	Attempt to move the queen so that it takes an opposition piece.	Typical	Try and drag and drop the queen to a vertical or horizontal or diagonal square from the start position when the squares to the end square are empty and the end square contains an opposition piece.	The game highlights the possible squares. When the user drops the piece in the legal square, the piece stays in the square and the piece originally there disappears. The move log is updated to show the move made in chess notation.	The game highlights the possible squares. When the user drops the piece in the legal square, the piece stays in the square and the piece originally there disappears. The move log is updated to show the move made in chess notation.	2:45
34	4di, 4diii	Attempt to perform an en passant when the opposition pawn has made 2 moves.	Erroneous	Try and drag and drop the pawn 1 square diagonally forwards when the opposition pawn that is directly next to it has made 2 moves.	The game does not highlight the diagonal square. When the user drops the piece in the square, the piece does not remain in the square and returns to the original square. The player can make another move.	The game does not highlight the diagonal square. When the user drops the piece in the square, the piece does not remain in the square and returns to the original square. The player can make another move.	2:57
35	4dii, 4diii	Attempt to perform an en passant when the opposition pawn movement was not the previous move.	Erroneous	Try and drag and drop the pawn 1 square diagonally forwards when the previous move was not the opposition pawn that is directly next to it.	The game does not highlight the diagonal square. When the user drops the piece in the square, the piece does not remain in the square and returns to the original square. The player can make another move.	The game does not highlight the diagonal square. When the user drops the piece in the square, the piece does not remain in the square and returns to the original square. The player can make another move.	3:09

37	4d	Attempt to perform an en passant when the opposition's previous move was a pawn by 2 squares directly next to the player's pawn	Typical	Try and drag and drop the pawn 1 square diagonally forwards when the previous move was the opposition pawn that is directly next to it and the opposition pawn's first move was 2 squares forward.	The game highlights the possible squares. When the user drops the pawn in the square, the pawn stays in the square and the opposition pawn disappears. The move log is updated to show the move made in chess notation.	The game highlights the possible squares. When the user drops the pawn in the square, the pawn stays in the square and the opposition pawn disappears. The move log is updated to show the move made in chess notation.	3:14
38	4ei	Attempt to castle when the squares between the king and rook are not empty.	Erroneous	Try to drag and drop the king 2 squares towards the player's rook when there is still a piece between them.	The game does not highlight the square two squares horizontal to the king. When the user drops the piece in the square, the king does not remain in the square and returns to the original square. The player can make another move.	The game does not highlight the square two squares horizontal to the king. When the user drops the piece in the square, the king does not remain in the square and returns to the original square. The player can make another move.	3:21 3:31
39	4eii	Attempt to castle after the king has moved.	Erroneous	Move the king back and forth. Try to drag and drop the king 2 squares towards the player's rook.	The game does not highlight the square two squares horizontal to the king. When the user drops the piece in the square, the king does not remain in the square and returns to the original square. The player can make another move.	The game does not highlight the square two squares horizontal to the king. When the user drops the piece in the square, the king does not remain in the square and returns to the original square. The player can make another move.	3:51
40	4eii	Attempt to castle when the rook has moved.	Erroneous	Move the rook back and forth. Try to drag and drop the king 2 squares towards the player's rook.	The game does not highlight the square two squares horizontal to the king. When the user drops the piece in the square, the king does not remain in the square and	The game does not highlight the square two squares horizontal to the king. When the user drops the piece in the square, the king does not remain in the square and	4:08

					returns to the original square. The player can make another move.	returns to the original square. The player can make another move.	
41	4eiii	Attempt to castle when the squares the king moves through are in check.	Erroneous	Get the opposition piece targeting the squares the king would move through. Try to drag and drop the king 2 squares towards the player's rook.	The game does not highlight the square two squares horizontal to the king. When the user drops the piece in the square, the king does not remain in the square and returns to the original square. The player can make another move.	The game does not highlight the square two squares horizontal to the king. When the user drops the piece in the square, the king does not remain in the square and returns to the original square. The player can make another move.	4:55
42	4eiv	Attempt to castle when the king is in check.	Erroneous	Get the opposition to put the king in check. Try to drag and drop the king 2 squares towards the player's rook.	The game does not highlight the square two squares horizontal to the king. When the user drops the piece in the square, the king does not remain in the square and returns to the original square. The player can make another move. The error message "Check" remains being displayed.	The game does not highlight the square two squares horizontal to the king. When the user drops the piece in the square, the king does not remain in the square and returns to the original square. The player can make another move. The error message "Check" remains being displayed.	5:10
43	4e	Attempt to king-side and queen-side castle when the king and the squares the king will move through are not in check and when both the rook and the king has not moved.	Typical	Try to drag and drop the king 2 squares towards the player's rook.	The game highlights the square two squares horizontal to the king. When the user drops the piece in the square, the king stays in the square and the position of the rook is updated to be positioned between the king's original position and the king's new position. The move log is	The game highlights the square two squares horizontal to the king. When the user drops the piece in the square, the king stays in the square and the position of the rook is updated to be positioned between the king's original position and the king's new position. The move log is	4:13 4:27

					updated to show the move made in chess notation.	updated to show the move made in chess notation.	
44	4f	Attempt to promote the pawn	Typical	Get the pawn to the 8th rank.	When the pawn gets to the 8th rank, the pawn is promoted to a queen. The move log is updated to show the move made in chess notation.	When the pawn gets to the 8th rank, the pawn is promoted to a queen. The move log is updated to show the move made in chess notation.	5:29 5:53
45	4g	Attempt to move the king into check	Erroneous	Try and move the king into a square which is under-threat by an opposition piece.	The game does not highlight the square that would put the king in check. When the user drops the king in the square, the king does not remain in the square and returns to the original square. The player can make another move.	The game does not highlight the square that would put the king in check. When the user drops the king in the square, the king does not remain in the square and returns to the original square. The player can make another move.	6:12
46	4h	Attempt to check the opposition king	Typical	Get in a position where the king could be captured.	The message “Check” is displayed. Only legal moves which resolves the check are displayed.	The message “Check” is displayed. Only legal moves which resolves the check are displayed.	6:18
47	4i	Attempt to move a piece which cannot stop the check when the player is in check.	Erroneous	Try and drag and drop a piece to a square which does not resolve the check.	The game does not highlight the square that would not stop the check. When the user drops the piece in the square, the piece does not remain in the square and returns to the original square. The player can make another move. The “Check” message is still displayed.	The game does not highlight the square that would not stop the check. When the user drops the piece in the square, the piece does not remain in the square and returns to the original square. The player can make another move. The “Check” message is still displayed.	6:36
48	4i	Attempt to take the piece which is causing the check	Typical	Drag and drop a piece to the square containing the piece causing the check.	The game highlights the square that would stop the check. When the user drops	The game highlights the square that would stop the check. When the user drops	6:24

					the piece in the square, the piece stays in the square and the opposition piece disappears. The move log is updated to show the move made in chess notation.	the piece in the square, the piece stays in the square and the opposition piece disappears. The move log is updated to show the move made in chess notation.	
49	4i	Attempt to block the piece which is causing the check	Typical	Drag and drop a piece to a square which is between the piece causing the check and the king.	The game highlights the square that would stop the check. When the user drops the piece in the square, the piece stays in the square. The move log is updated to show the move made in chess notation.	The game highlights the square that would stop the check. When the user drops the piece in the square, the piece stays in the square. The move log is updated to show the move made in chess notation.	6:37
50	4j	Attempt to make a checkmate	Typical	Get in a position where the user has no legal moves and is in check.	Message displayed saying “Checkmate”. The user cannot make another move.	Message displayed saying “Checkmate”. The user cannot make another move.	6:47
51	5d	Attempt to undo a move	Typical	Click the “Undo Move” Button	The previous move is undone. The piece returns to the original location. The move log is updated removing the previously played move. The move before the previous move is displayed on the board.	The previous move is undone. The piece returns to the original location. The move log is updated removing the previously played move. The move before the previous move is displayed on the board.	6:55
52	4k	Attempt to make a stalemate	Typical	Get in a position where the user has no legal moves but is not in check.	Message displayed saying “Stalemate”. The user cannot make another move.	Message displayed saying “Stalemate”. The user cannot make another move.	7:18
53	4l	Attempt to draw by repetition.	Typical	Repeat the same two moves of moves 3 times.	Message displayed saying “Draw by repetition”. The user cannot make another move.	Message displayed saying “Draw by repetition”. The user cannot make another move.	7:32
54	7a	Attempt to save the game without entering a name.	Erroneous	Name: “” Click the Save Button	Error Message: “Enter a valid name”	Error Message: “Enter a valid name”	7:43

					Game is not saved.	Game is not saved.	
55	7a	Attempt to save the game after entering a name.	Typical	Name: “Test Game 1” Click the Save Button	The game is saved to the server. The user is directed back to the Main Menu screen.	The game is saved to the server. The user is directed back to the Main Menu screen.	7:45
56	8a	Attempt to select a puzzle range which contains no puzzles.	Erroneous	Minimum rating: 0 Maximum rating: 0 Click the button “Play”	A label is displayed saying “There are no puzzles within the selected range”.	A label is displayed saying “There are no puzzles within the selected range”.	7:53
57	8a, 8b, 8c	Attempt to load into a puzzle	Typical	Minimum rating: 1593 Maximum rating: 1600 Click the button “Play”	A puzzle within the range is loaded, and the chess position of the puzzle is displayed on the board. The previous move made by the opposition is played. The user can make a move.	A puzzle within the range is loaded, and the chess position of the puzzle is displayed on the board. The previous move made by the opposition is played. The user can make a move.	8:07
58	8d	Attempt to incorrectly complete the puzzles.	Erroneous	Drag and drop a piece to a legal but incorrect move	When an incorrect move is played, a “Incorrect” label is displayed. The piece is returned to its original position. The user can make another move.	When an incorrect move is played, a “Incorrect” label is displayed. The piece is returned to its original position. The user can make another move.	8:17
59	8d	Attempt to correctly complete the puzzle.	Typical	Drag and drop a piece to a legal but correct move	When a correct move is played, a “Correct” label is displayed. When the user completes the puzzle, the user is taken to another puzzle within the same range. The next puzzle is of the same rating or higher than the previous puzzle.	When a correct move is played, a “Correct” label is displayed. When the user completes the puzzle, the user is taken to another puzzle within the same range. The next puzzle is of the same rating or higher than the previous puzzle.	8:29
60		Attempt to complete all the puzzles within a range.	Erroneous	Complete the final puzzle within a range.	The user is directed to the Puzzle Selector Menu to reselect a new puzzle range.	The user is directed to the Puzzle Selector Menu to reselect a new puzzle range.	8:36

Test Video 2

61	6a,6b	Attempt to play against a 1600 rated (Advanced) AI on Chess.com using my “Hard” Difficulty AI. My AI playing as White.	Typical	Input the move made by the Chess.com AI into my AI. For every move produced by my AI, I input the move into the Chess.com AI.	The games between the AI’s should be a close.	My AI beat the Chess.com Advanced AI. The response times were similar/slightly slower than Chess.com	Good Moves: 0:02 0:18 0:35 1:05 1:38 2:09
62		Attempt to play against a 1200 rated (Intermediate) AI on Chess.com using my “Intermediate” Difficulty AI. My AI playing as Black.				My AI beat the Chess.com Intermediate AI. The response times were like Chess.com	Good Moves: 3:22 3:28 3:34 3:51 4:43
63		Attempt to play against a 850 rated (Beginner) AI on Chess.com using my “Easy” Difficulty AI. My AI playing as Black.				My AI beat the Chess.com Beginner AI. The response times were like Chess.com	Good Moves: 5:11 5:26 5:52 6:36
64	7b	Attempt to filter for the games previously played by “Colour Played As”.	Typical	Colour You Played As: “Black” Result: “Win”, “Loss”, “Draw” Game Against: “Both”	The game that will appear: “Game Against 1600 Rated Bot”	The game that will appear: “Game Against 1600 Rated Bot”	7:06

65	7b	Attempt to filter for the games previously played by “Colour Played As”.	Typical	Colour You Played As: “White” Result: “Win”, “Loss”, “Draw” Game Against: “Both”	The games that will appear: “Game Against 850 Rated Bot” “Game Against 1200 Rated Bot” “Test Game 1”	The games that will appear: “Game Against 850 Rated Bot” “Game Against 1200 Rated Bot” “Test Game 1”	7:05
66	7b	Attempt to filter where the user has not selected a Result.	Erroneous	Deselect All Results in Drop Down Menu. Then Set Filter.	No games will be displayed	No games will be displayed	7:11
67	7b	Attempt to filter for the games previously played by “Played Against”.	Typical	Colour You Played As: “Both” Result: “Win”, Loss” Game Against: “Human”	The games that will appear: “Test Game 1”	The games that will appear: “Test Game 1”	7:28
68	7b	Attempt to filter for the games previously played by “Played Against”.	Typical	Colour You Played As: “Both” Result: “Win”, Loss” Game Against: “AI”	The games that will appear: “Game Against 850 Rated Bot” “Game Against 1200 Rated Bot” “Game Against 1600 Rated Bot”	The games that will appear: “Game Against 850 Rated Bot” “Game Against 1200 Rated Bot” “Game Against 1600 Rated Bot”	7:28
69	7c	Attempt to review the same game.	Typical	Select the Game “Test Game 1” Click the “Load Game” button.	User is directed to the Game Review Menu. The chess pieces are set up in the starting position on the board.	User is directed to the Game Review Menu. The chess pieces are set up in the starting position on the board.	7:39
70	7d	Attempt to progress through the saved game.	Typical	Click the “Make Move"	When the “Make Move” button is clicked, the game plays the next move in the saved game.	When the “Make Move” button is clicked, the game plays the next move in the saved game.	7:40
71	7d	Attempt to undo moves in the saved game.	Typical	Click the “Undo Move” button.	When the “Undo Move” button is clicked, the game undoes the previous move in the saved game.	When the “Undo Move” button is clicked, the game undoes the previous move in the saved game.	7:43

72	7e	Attempt to continue the game from a position within the game.	Typical	Click the “Continue Game” button.	When the “Continue Game” button is clicked, the game directs the user to the Game Menu. The user then can make new moves within the game.	When the “Continue Game” button is clicked, the game directs the user to the Game Menu. The user then can make new moves within the game.	7:46
73	7e	Attempt to make new moves.	Typical	Drag and drop the pieces of the player to move.	The user can pick up the piece. When the move is made, the move made is added to the move log.	The user can pick up the piece. When the move is made, the move made is added to the move log.	7:50
74	7a	Attempt to save the variation of the game.	Typical	Name: “Test Game 1 Variation” Click the Save Button	The game is saved to the server. The user is directed back to the Game Review screen.	The game is saved to the server. The user is directed back to the Game Review screen.	7:56
75	7b	Attempt to filter for the new game variation.	Typical	Colour You Played As: “Both” Result: “Win”, Loss” Game Against: “Both”	“Test Game 1 Variation” should appear.	“Test Game 1 Variation” should appear.	8:11

Evaluation

Personal evaluation

1 - The system must allow users to register for an account.

I was able to successfully complete this objective. The chess program allows users to create an account. The account details are validated. The system checks if the username is unique, password has a special character and the password is at least 8 characters long. If any of these conditions are not met, when the “Submit” button is pressed a corresponding error message is displayed to the user.

Instead of using an auto-incrementing userID as the primary key in the user table, instead it would be more efficient to make the username the primary key as the username must be unique anyway.

2 - The system must allow users to log-in to their account.

I was able to successfully complete this objective. When logging in, users must enter the correct email and password, else a corresponding error message is displayed to alert the users to input the valid details. The program verifies whether the username belongs to an account by checking whether there is an entry for that username in the database. The passwords are stored securely on in the database. The process of encryption works by hashing the inputted password with the salt on the client device and it compares the hashed password to the hashed password is stored in the database when the account was created.

3 - The system must allow users to navigate the system.

I was able to successfully complete this objective. The application has menus which the user can navigate between. The user interface is intuitive as all drop down menu, text inputs and selectors have text in front to describe their purpose. In addition, I tried to reduce the number of menus in my system to allow it to be easy to use for all years in my school. All buttons in my system conducts its intended function by directing users to the corresponding menu or executing the tasks which is described by the label. My objectives did not include all the buttons in the system, as I did not conceive the number of buttons required.

4 - The system must follow the rules of chess.

I was able to successfully complete this objective. Within my testing video, I have included tests for all the game rules. The system ensures that the users take turn in making moves starting with white moving first. I have tested to ensure that all the piece moves in their set ways. I also tested to ensure, I have programmed all the special moves like castling, en passant and pawn promotion. I have performed a Perft test which is an essential tool used when debugging. It tests for all the legal moves that can be carried out to a certain depth. I have tested whether the application correctly responds to Checks, Checkmates, Stalemates and Draws.

One improvement that I can make to my system is the ability for users select which piece the pawn promotes to. However, in normal play, users typically always default to picking a queen.

4 - The system must allow users to be able to play chess.

I was able to successfully complete this objective. When playing chess on the application, the user-interface is updated to display the chess pieces. The application allows users to interact with the pieces by dragging and dropping the pieces from and to their desired square. I have ensured the

robustness of the system by validating the moves the user can try to make so that no errors appear. I have included a move record feature which writes the moves made in chess notation which is updated after each move has been made. The undo button works and maintains the ability for castling and en passant rights for the different moves.

One improvement that I can make to my system is the ability to see all the pieces that have been taken on the side of the board. As after a player takes another piece, the taken piece just disappears from the board.

5 - The system must allow users to play against an AI.

I was able to successfully complete this objective. The user can play against an AI of a range difficulties from “Easy” to “Hard”. There is a noticeable difference in difficulty between difficulties, therefore providing users of a range of skill levels to be challenged. The “Easy” AI provides users of around 750 rated to be challenged. A 750 rated chess player is typically seen as a below average player, therefore a good challenge for a beginner. The “Intermediate” AI provides users of around 1200 rated to be challenged. A 1200 rated chess player is typically seen as an average player, therefore a good challenge for most people. The “Hard” AI provides users of around 1600 rated to be challenged. A 1600 rated chess player is seen as above average, therefore a good challenge for a strong player.

Whilst the response times for “Easy”, “Intermediate” and “Hard” shown in the testing video provides similar response times to that of chess.com, I was limited to a search depth of 4 as beyond a depth of 4 the response times became too slow and so un-useable. Therefore, if I want to further increase the range of players the AI can beat, I must further optimise the chess engine to allow for a max search depth greater than 4. In addition, on older computers, there is a noticeable delay in the time it takes the AI to make a move on “Hard” difficulty.

6 – The system must provide users the ability to review previously played games.

I was able to successfully complete this objective. The application allows users to save games that they have played to a server. The application then allows users to retrieve and review the game that they played whenever through logging into their account. Then they can complete their game or try different moves than the one they played, either against an AI or by themselves.

7 - The system must provide users the ability to complete chess puzzles.

I was able to successfully complete this objective. The application selects a puzzle within the ratings selected by the user. The application sets up the board using the FEN notation of the board stored on the server. Users are notified with an error message when the inputted move is incorrect. Once the user makes the correct sequence of moves the user to directed to another puzzle and the user does not complete the same puzzle twice.

End User Evaluation

I spoke again to Ethan, who I interviewed at the beginning of the project, where I gave him my chess application. He used it, creating logins, and played games against the AI, completed puzzles and reviewing games. He emailed back to me with the following feedback:

Upon testing the AI function of the chess application, I was pleasantly surprised to find its ability easily matched or exceeded my own. Moreover, I was able to customise the difficulty to my liking which was particularly helpful in finding an opponent that was both challenging but also possible to beat. The engine was also able to see a variety of tactics such as pins, sacrifices and skewers which made the game feel especially realistic. However, I did it find it mildly frustrating that it only was able to use a single opening which failed to stimulate the realities of playing against a real opponent. However, overall the application seemed very effective playing similarly to competitors such as chess.com and having equivalent functionality.

Next, I evaluated the game review function which allowed me to look back at previously saved games. The ability to save games with custom names and then filter to find those games is extremely useful allowing games to be dated and looked back upon that could be relevant to upcoming events or opponents. However, the game review function did fail to provide analysis which though not affecting me personally it could affect many people who would have to use outside analysis which could pose an inconvenience.

Finally, I tested the puzzle function which offered a wide variety of different puzzles ranging from 0-2000 Elo in difficulty. Though this range is substantial, and the tactics involved in the puzzles also extremely varied I did myself wanting the ability to play against higher rated puzzles which weren't available. However overall the puzzle function was highly effective and placed me into many very relevant positions with crucial decisions which helped me to train my ability to spot important tactics.

Overall the application offers a good range of functions with its own Chess AI arguably being the best part of the program. It's ability to change in difficulty to match the ability of its opponent making it relevant to all beginner and intermediate level players to practice with. Though it may have a few downfalls as noted before these are truly minor and don't take away from the overall user experience which on the whole is extremely positive and beneficial for both leisure and competitive use.

Comments on Feedback

The feedback from Ethan shows that I have addressed all the objectives, and that the system can be used in the way it was intended. However, Ethan did highlight area of improvements for the application.

Whilst Ethan wrote about how he thought the AI provided a satisfactory level of challenge and awareness to different chess tactics, he wrote about how he would like to see the AI perform a range of different openings, allowing for a wide variety of games to prevent repeating games.

For the game review section, Ethan wrote about how I met the objective he set out for me in the initial interview, when talking about an account system, he said, “An account system would be necessary as it would allow me to track my progress and would allow me to review the game that I have played”. However, in the email above he further added an idea of improvement of adding my own natural language chatbot that provides analysis on each move. Whilst this feature would be nice to implement in the future, I feel like the feature is far to complex to program myself.

For the puzzle section, Ethan wrote about how the puzzles are beneficial as it exposed him to lot of different relevant positions where critical decisions must be made. Therefore, it allowed him to improve his pattern recognition for different tactics in different chess positions.

Improvements

If I had more time, I would like to add the following features and improvements:

- Further optimisation of the chess engine to allow for a higher maximum depth to be reached, therefore a stronger engine. The way I could further optimise the engine is by using a transposition table. A transposition table is a hash table which stores the positions previously searched, how deep the search was and the evaluation of the board. This reduce the number of moves pruned and improve my move ordering function.
- Include the ability for users to choose which piece the pawn promotes to. The way I could implement it is by adding a pop menu when the pawn reaches the 8^{th} rank. This pop up can be used to allow users to choose which pawn the promotes to.
- Include a taken piece section when playing a game, where instead of removing the rect from the screen, it moves the rect to the side of the board.
- Include an inbuilt natural language chat bot in the game review section, where the application looks at the position and the application tell the user why a move is good/bad.
- Include an online multiplayer system which allows users to play over a network. The way I could implement is by hosting the application on a central server where users can use their device to connect and play.
- Include a variety of different openings the chess AI can play. The way I could implement this feature is by downloading a database of chess openings where the application randomly selects a set of moves to play at the beginning of the game.
- Increase the range of puzzles available for the users. The way I could implement this feature is by downloading a database of chess puzzles of a greater range and add it to my database.

Overall, I think that I have met all the objectives set out at the beginning of the project. However, there are many ways the system can be extended to help further to help aid the development of chess players.

References

Implementation of Chess AI's

- https://towardsdatascience.com/dissecting-stockfish-part-2-in-depth-look-at-a-chess-engine-2643cdc35c9a
- https://www.chessprogramming.org

Rules of Chess

- https://www.dicebreaker.com/games/chess/how-to/how-to-play-chess

Chess Notation

- https://www.chesshouse.com/blogs/education/how-to-read-and-write-algebraic-chess-notation