CHESS AI
OSCAR HOGBEN

GODALMING COLLEGE
oscarhogben.co.uk

Contents

Analysis

Current Chess Game

Identification of problem

Chess can be played in multiple different ways: online, on a board, in the post and many more, but you always need another person to play with. This restriction stops chess players from being able to play whenever they want and to practice for their competitive games.

The end user I have chosen is Will Ward who is a passionate chess player who finds this an issue when he wants to play chess.

The solution to this problem that I will be investigating is a chess AI where the player can play against the computer.

Interview with Chess Player

Interviewee: Will Ward
Occupation (in context): Chess Player

Q: Do you find needing another player a restriction with chess?
A: Yes it's definitely a restriction but it's nice to have a social element to the game.

Q: Would you play more chess if you didn't need someone else?
A: Yes I would especially to practice for games against real people. Also, it would be great to have the freedom of being able to play at any time.

Q: Do you think a computer AI would be an adequate solution to this restriction?
A: Yes as long as the computer works well and feels like it could be a player.

Q: What would you expect a computer AI chess game to look like?
A: It would be great to have a GUI, but I don't really mind if all the menus, board and pieces are easy to read and understand.

Q: How would you prefer to enter your moves in the computer?
A: If there was a GUI you could click the piece you want to move then the place you want to move it to, but it would be fine if you were just entering the coordinates.

Q: Would you prefer if there were different difficulty levels for the chess AI, and if so, how many?
A: Different difficulties would be very important to progress when you're practicing and training yourself and having a system that allows as many difficulty levels as possible would be even better because you can make small changes.

Q: The chess AI is not likely to make moves instantly as it needs time to process. At what point would you consider the wait time to be too long?
A: If it doesn't take longer than a player that would be ok. So nothing more than one minute.

Q: For a chess AI game, would it be useful to have the previous board state saved so you can go back to it after terminating the program?
A: Yes that would be very handy.

Interview Summary:
A chess AI would be a welcome solution to the two-player restriction and would certainly be used. The computer game should be easy to understand for any chess player and can be played with the entry of coordinates. An extensive array of difficulties would also be appreciated to suit every level of player. The AI should also try to be as realistic as a player by not taking more than a player takes (about 1 min) to make a move. It is also clear that it would be very useful if the chess game automatically saved and could be loaded again upon running the program.

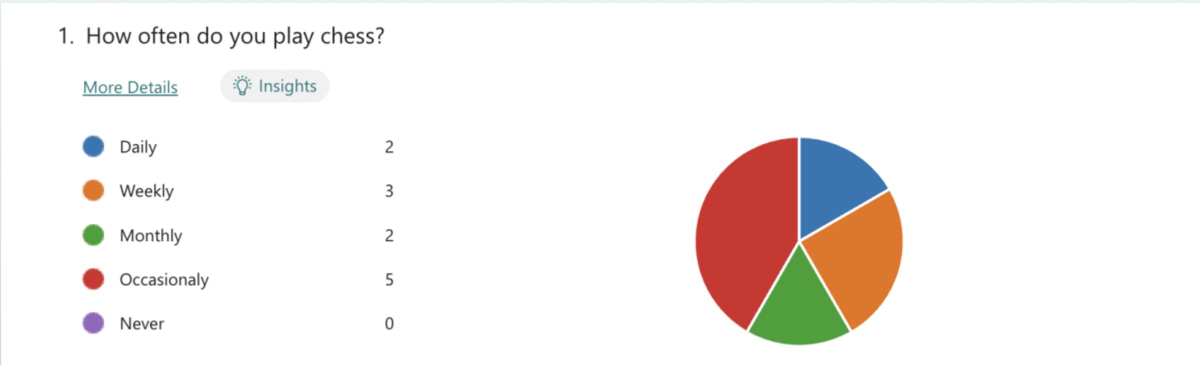
Questionnaire

https://forms.office.com/e/HsdjF5mQEc

Q1

1. How often do you play chess? *

- Daily
- Weekly
- Monthly
- Occasionaly
- Never



This question was intended to ensure the validity of our responses by checking the participants played chess. This was 100% successful as everyone plays chess.

Q2

2. Do you have trouble finding someone to play with?

◯ Yes

◯ No

2. Do you have trouble finding someone to play with?

More Details   Insights

Yes	10
No	2

This question was intended to clarify the problem of finding someone else to play chess with you. It found 83% of people to have trouble finding someone. This is evidence to support that chess needing 2 player is a limitation.

Q3

3. Would you play more chess if you didn't need another person to play with?

◯ Yes

◯ No

3. Would you play more chess if you didn't need another person to play with?

More Details   Insights

Yes	12
No	0

This question was to further investigate and support the argument that chess would be played more if it didn't need 2 people. 100% of people said they would play more chess if they didn't need another person.

Q4

4. One solution to needing 2 players, is for the second player to be a computer controlled AI (Artificial Intelligence). Can you think of any other solutions?

Enter your answer

This question was designed to look for and compare solutions to the 2-player requirement. 7/12 people answered this question, and all of the responses were either play by yourself or join an (online) club. Playing by yourself would provide a way to play whenever you want with just one player, however it does remove the competitive aspect of the game. Joining an online club is a solution to not finding people to play with however it does still mean you have to organise a game with a player which means you can't play exactly when you want to.

Q5

5. If it was available, would you play against an AI?

- Yes
- No

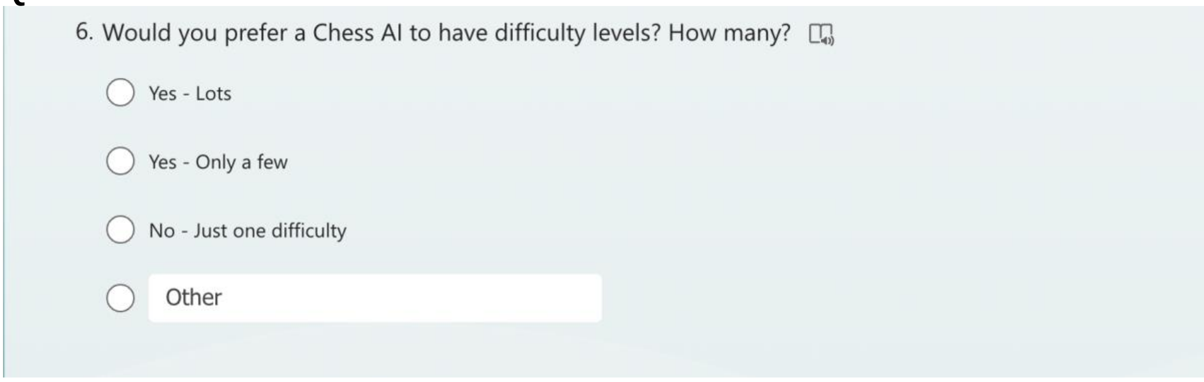
5. If it was available, would you play against an AI?

More Details

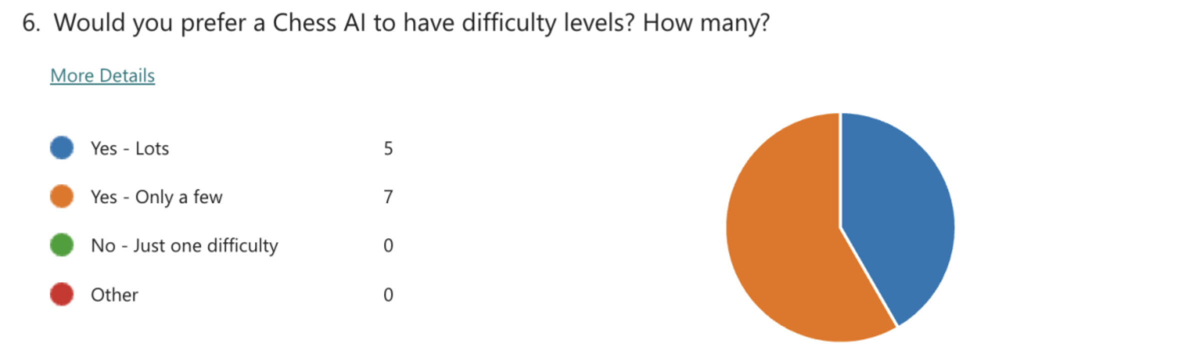
Yes	12
No	0

This question was designed to gauge the usefulness of a chess AI as a solution to the problem. It showed 100% of people saying they would play against the AI which strongly suggests the AI would be used.

Q6

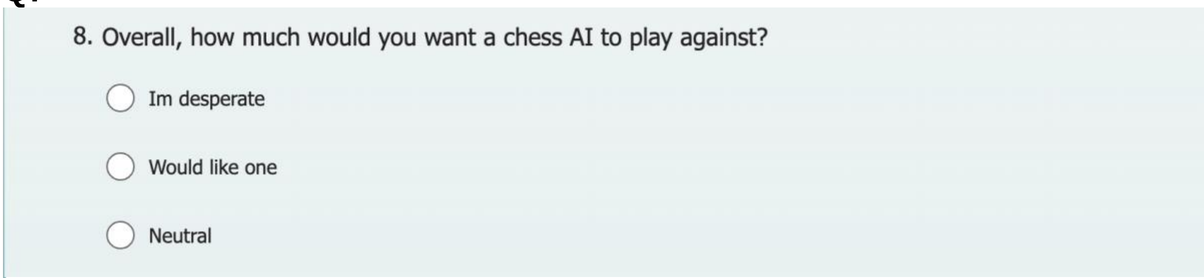

6. Would you prefer a Chess AI to have difficulty levels? How many?

- Yes - Lots
- Yes - Only a few
- No - Just one difficulty
- Other



This question was to get an idea on weather people wanted custom difficulty and, if so, how many difficulty levels they waned. 100% of people wanted custom levels of which 58% wanted only a few and the other 42% wanted lots.

Q7


8. Overall, how much would you want a chess AI to play against?

- Im desperate
- Would like one
- Neutral

This question was to get an idea of the demand for an AI as a solution. 100% of people expressed interest and 50% of people expressed strong interest.

Current Systems and Methods Used

The current main chess system is to play with a physical board and pieces with another player in person. This requires 2 players that want to play with each other and are in the same place at the same time. There are multiple solutions to this already in place:

Correspondence Chess

Correspondence chess is a form of long-distance chess traditionally through the post but is most common through the internet now. It was created and used to solve the restriction of needing another player in the same place as you at the same time.

It works by sending a chess games move history to a second player in a different location. After receiving the chess moves, the second player will recreate the current board state using the move history, make their move, add it to the move history and send it back to the first player for them to do the same.

https://en.wikipedia.org/wiki/Correspondence_chess

Correspondence chess game tends to take much longer as the moves have to be re-sent after every single move and you would have to wait for your opponent to have time to make their move. This does also mean that you have much longer to make each move which reduces error and usually means you play better.

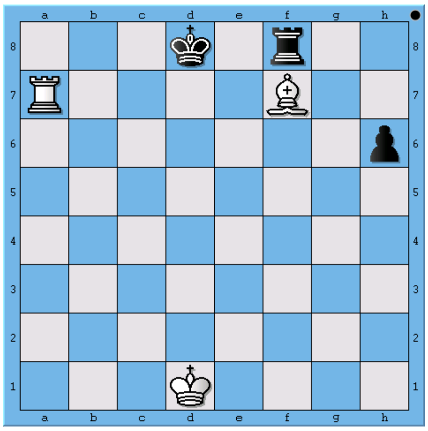
Correspondence chess via post began to die out as people started using the internet for it instead as its cheaper, quicker and more reliable.

https://www.chess.com/terms/correspondence-chess

Chess Puzzles

Chess puzzles are scenarios that are pre-made and only include a few pieces. The puzzles are also sometimes on smaller and more restrictive boards. There is always an objective to the puzzles which is usually to get your opponent's king into checkmate in a limited number of moves. When playing chess puzzles, you should always assume the opponent will play the best moves possible and your method should provide a definite win.

The image opposite is an example of a chess puzzle in which the player white and their objective to get black's king into checkmate in a limited number of moves.



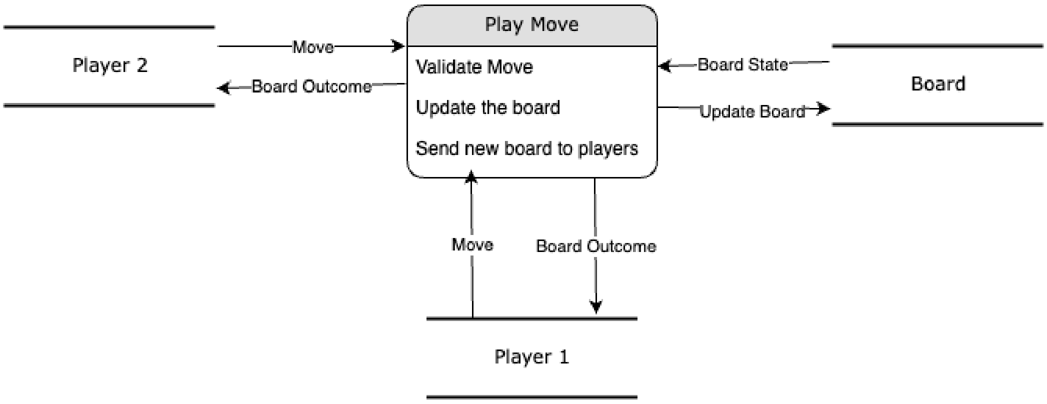
https://www.quora.com/Do-chess-puzzles-improve-your-skill-at-chess

There are other types of chess puzzle for example to survive a number of moves without your having your king in checkmate.

Chess puzzles are traditionally found in newspapers however there are now lots of sites online where you can find a whole range of puzzles that are also controlled by AI.

Data Flow Diagram

This Data flow Diagram shows what information is transported and where it is transported to and from.



Data Dictionary – Computer Based

This is a data dictionary showing each data item in the chess game, its type, its frequency, and the set of data it contains.

Item	Type	Frequency	Set
Square	Pointer	64	Piece / blank
Piece	Object	32	KQBNRP
Taken pieces	Piece	0 - 32	KQBNRP
Board	2D Array (matrix)	1	Square
Previous Moves	Array	∞	Move
Move	String	∞	Notation for move
Black Piece List	Array	16	Black Pieces
White Piece List	Array	16	White Pieces

IPSO Chart - Move

This is an IPSO Chart showing the process of a player making a move.

Input Move Board Player (colour) White/Black Piece Array Pieces	**Output** Board Move History (if requested)
Process Move Validation Amend Board Add move to move history Edit Piece Properties Check for Check and Checkmate Check for Stalemate Amend white/black piece array	**Storage** Board Move History Game State (check/checkmate/stalemate) Board Piece(s) White/black piece array

IPSO Chart – Enter Move Coordinates

This is an IPSO chart showing the process of a player entering coordinates.

Input Piece-to-move coordinates Position to move to coordinate	**Output** Invalid Coordinates (if coordinates are invalid) Valid Coordinates
Process Validate coordinates are in range Validate the piece belongs to the player	**Storage** Current Move Coordinates

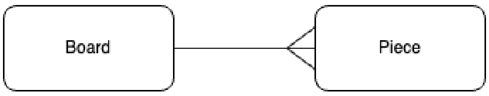
IPSO Chart – Check for Check, Checkmate and Stalemate

This is an IPSO chart showing the check for Check, Checkmate and Stalemate.

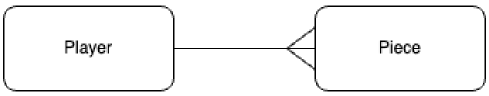
Input Board White and Black Piece arrays List of Valid Moves Current one-piece-left move count	**Output** Game Status (check, checkmate, stalemate) One-piece-left count (if needed)
Process Check for the exitance of valid moves Check one-piece-left move count < 50 Check valid moves: if any cause check Check valid moves: if any take out of check Increment one-piece-left count if needed	**Storage** Game Status (check, checkmate, stalemate) One-piece-left count (if needed)

Entity Relationship Diagram's

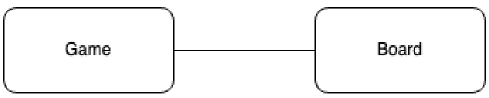
This entity relationship diagram shows the one-to-many relationship between the board and the pieces. One board can have lots of pieces, but one piece only has one board.



This entity relationship diagram shows the one-to-many relationship between the players and the pieces. One player owns many pieces, but one piece belongs to one player.

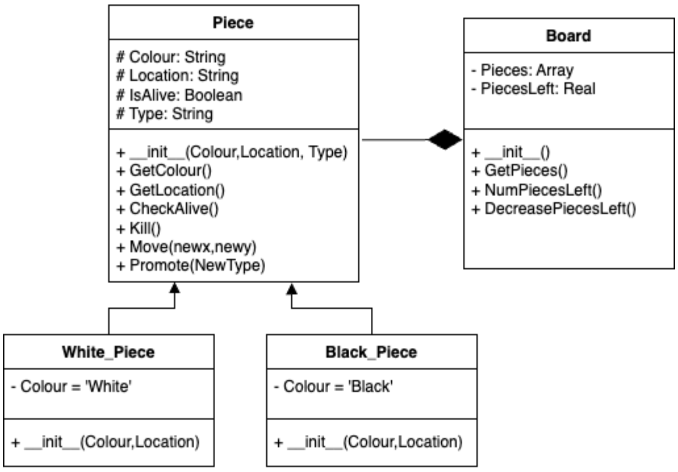


This entity relationship diagram shows the one-to-one relationship between a game and a board. Each game has one board and each board belongs to one game.



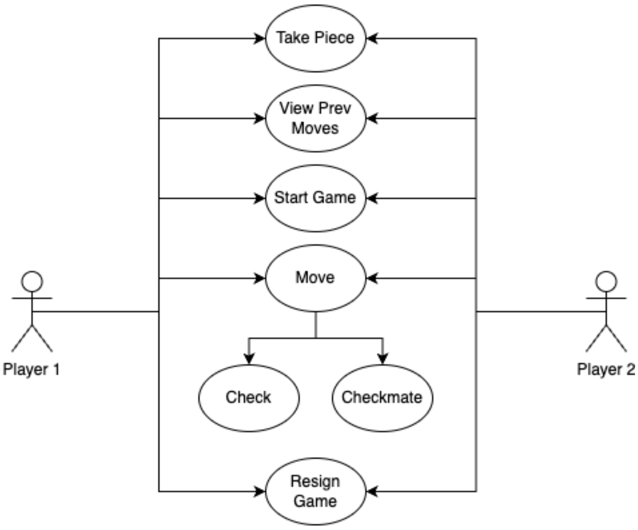
UML Diagram

This UML Diagram shows the relationship between the board, pieces, and coloured pieces. It shows the coloured pieces inherit from piece and the board is composed of pieces.



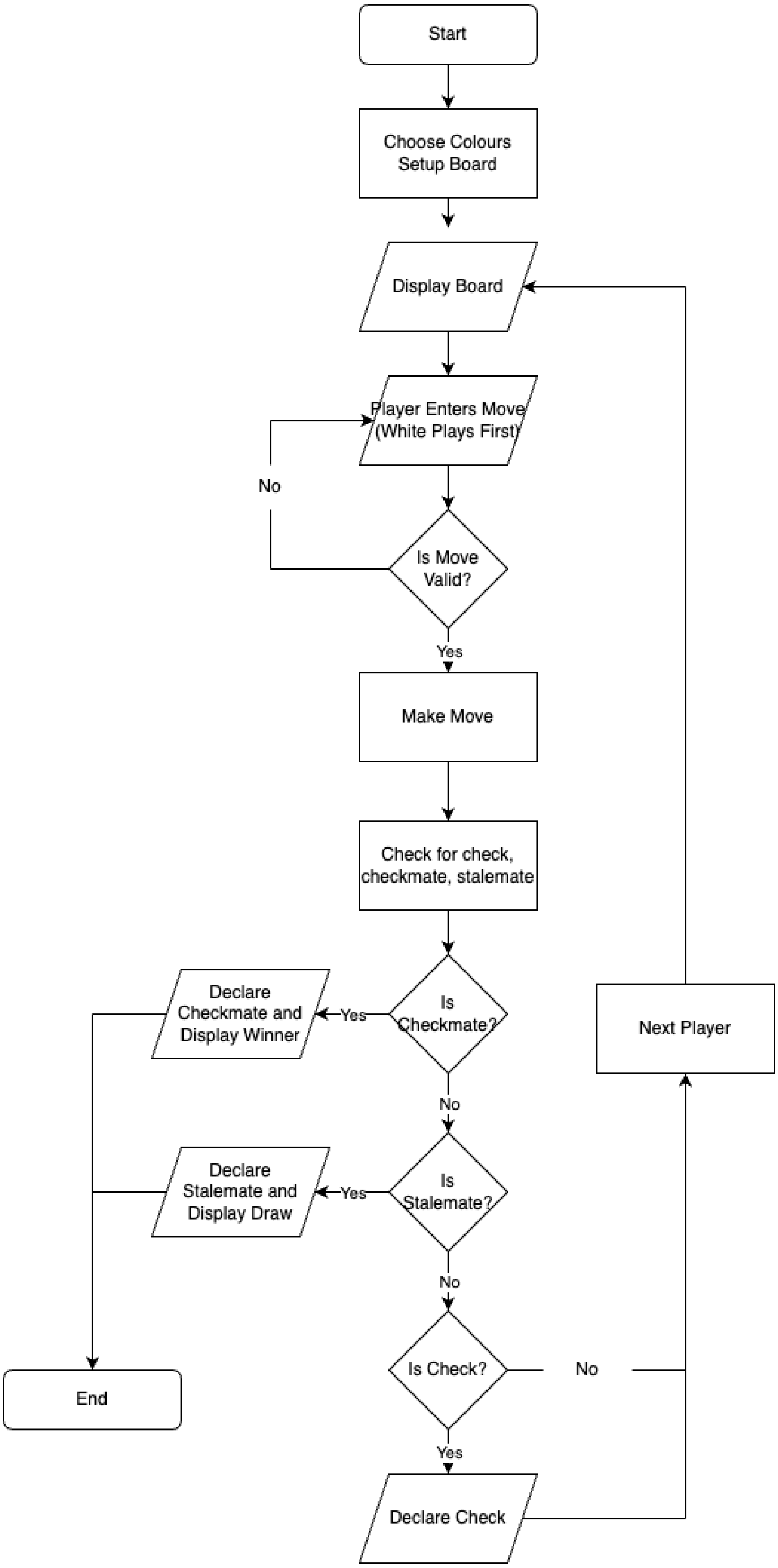
Use Case Diagram

This Use Case Diagram shows how each player can interact with different features and functions in the chess game with two players: Player 1, Player 2.



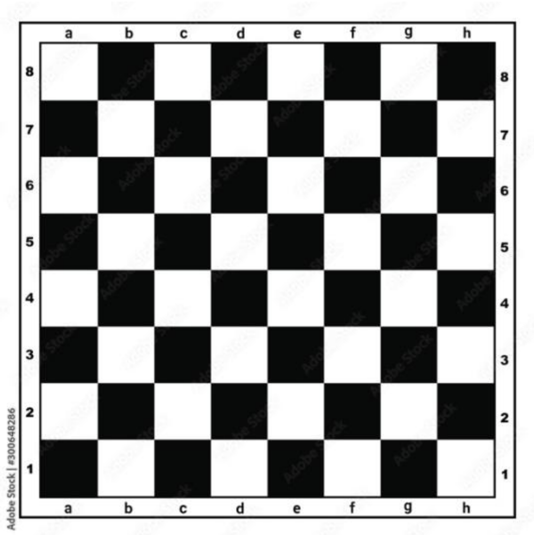
Flowchart

This flowchart shows how a whole chess game operates from start to finish.



The Board

Chess is a two-player board game that is deeply based on strategy. It is played using one board which is an 8x8 grid made up of black and white squares. Each colour alternates every other square. This also makes the diagonals of any square the same colour as the square.



- image from: stock.adobe.com

There is also ascending numbers on the left of the board increasing from bottom to top (starting at 1 ending at 8) in the vertical direction, while there are letters in order of the alphabet in increasing from left to right (starting at 'a' ending at 'h') in the horizontal direction.
The combination of these letters and numbers provides unique coordinates of every board square according to which letter and number the board square lines up with in the horizontal and vertical directions.
The letter will always be shown before the number in the coordinate. For example, a1 would be the furthest bottom left square on the board and h8 would be the furthest top right square on the board.

Each vertical line on the chess board is called a 'file' and each horizontal line is called a rank.

Each player will sit opposite each other on the board, one sitting with the coordinate numbers ascending away from them while the other has them ascending towards them.

-Info from: https://www.chess.com/article/view/how-to-set-up-a-chessboard

Pieces + Piece Layout

In chess there are 6 unique pieces each of which can be black or white according to which player the piece belongs. Each piece has a unique method of moving on the board which is described by a series of rules that are mostly individual to the piece.

A piece can only move to a square if the square does not have any piece of the same colour on it.

Pawn:
The standard pawn looks like this:

https://chess.fandom.com/wiki/Pawn

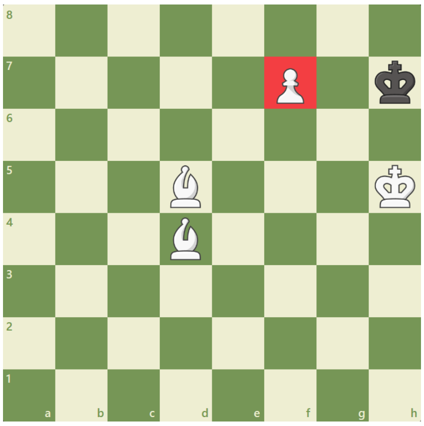
A pawn is the most common and least powerful piece on the board. In the chess points system, it's worth 1 point. Each player has 8 at the start of the game which makes 16 pawns in a chess set 8 of which are white and the other 8 of which are black.

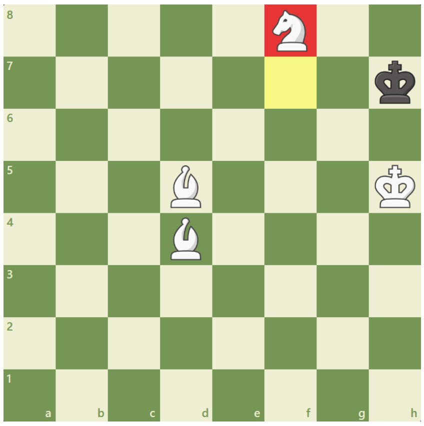
A Pawn can move forward one square if there is no other piece Infront of it (relative to the owning players position) or pawn it can move two squares forward if both squares are empty and it is that pawns first move.

It can move diagonal forward to the left or right if there is a piece that belongs to the opposing player (is the opposite colour) on the square being moved to.

The pawn is also subject to En Passant (page 21).

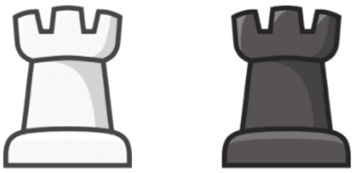
If the pawn makes it to the other end of the board to where is started, it can 'promote' itself. This allows the pawn to be switched out for a knight, bishop, rook or queen. This happens during the move of the pawn making it to the other end and it is the owning players choice as to what the pawn promotes into.





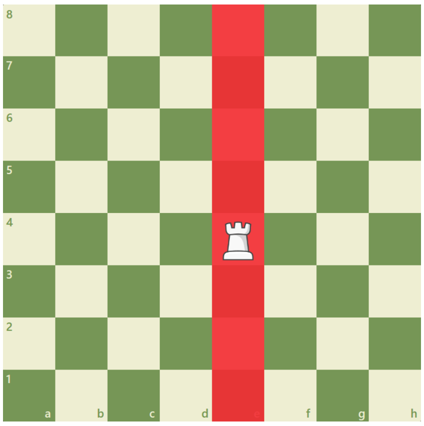
https://www.chess.com/terms/chess-pawn

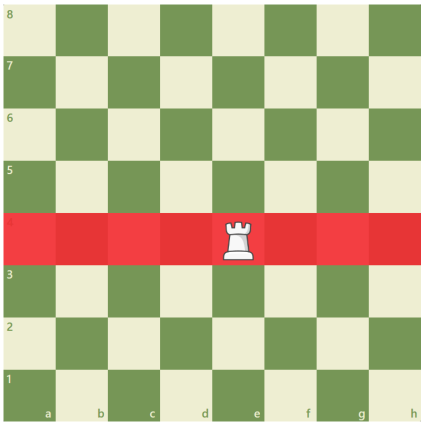
Rook:
The standard rook looks like this:



https://chess.fandom.com/wiki/Rook

A rook is the second most powerful piece. It is worth 5 points. It can move forward, backward or sideways, but cannot move diagonally (like a queen or a bishop). The rook can move up or down vertically on any file or left or right horizontally on any rank.





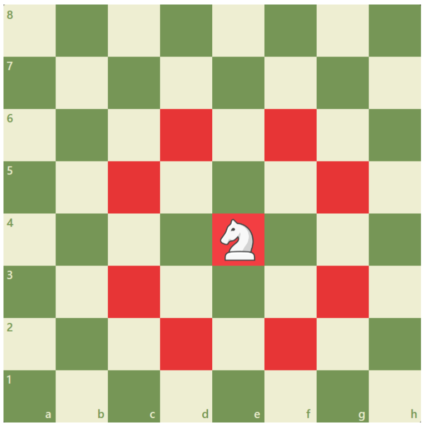
The rook cannot move to a location in which its path is blocked by any other pieces.

https://www.chess.com/terms/chess-rook

Knight:
The standard knight looks like this:

 https://chess.fandom.com/wiki/Knight

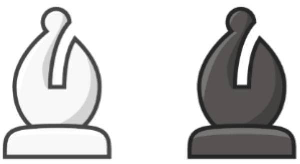
The knight moves multiple squares each move. Its worth 3 points. It either moves up or down one square vertically and over two squares horizontally OR up or down two squares vertically and over one square horizontally. In the diagram you can see all the potential knight moves.



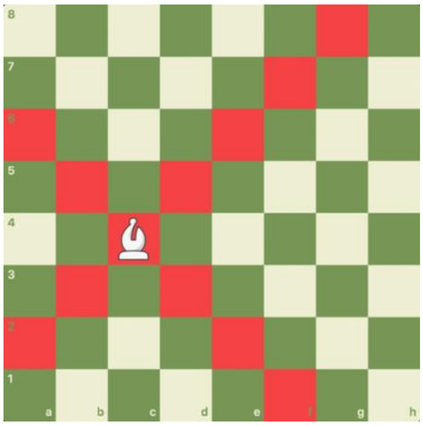
The knight is the only piece that can jump over any other piece to get to its location. It can still move to its location if there are pieces between it and the location.

https://www.chess.com/terms/chess-knight

Bishop:
The standard bishop looks like this:

 https://chess.fandom.com/wiki/Bishop

The bishop moves only in diagonals. Its worth 3 points. They can move to any square diagonal to them on the board only if there is no piece blocking their path. This means one bishop can only reach half the board.

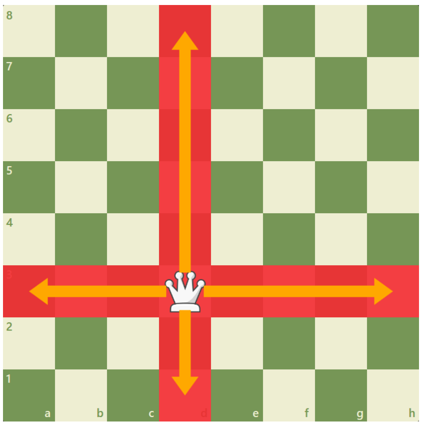


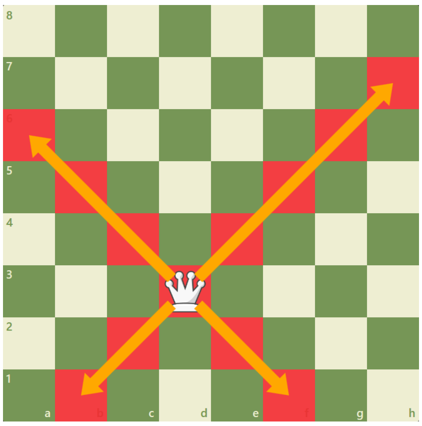
https://www.chess.com/terms/chess-bishop

Queen:
The standard queen looks like this:

https://chess.fandom.com/wiki/Queen

The queen is the most powerful piece. It’s worth 9 points. The queen can move the same way a rook can, moving freely up and down on any file and left and right on any rank. Or queen can also move freely on any diagonal like a bishop.



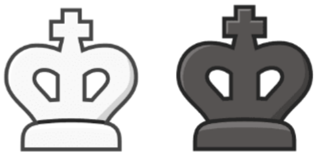


The queen cannot move anywhere that is blocked by another piece.

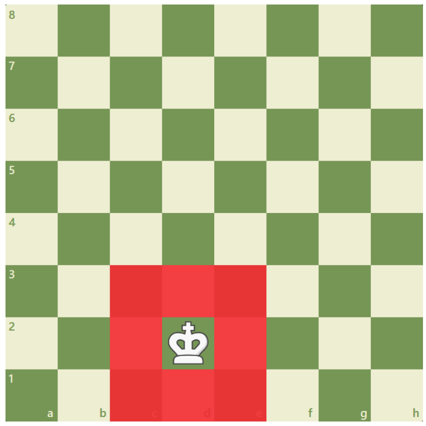
https://www.chess.com/terms/chess-queen

King:

The standard king looks like this:

 https://chess.fandom.com/wiki/King

The king is the most important piece in chess. If the king is taken by the other player, the game ends. The king can only move one square in any direction.



https://www.chess.com/terms/chess-king

Piece Layout:

The board should be oriented so the light (white) square is in the bottom right corner. A player will be positioned at the top of the board, and the other at the bottom. The player at the top of the board should use the black pieces, and the player at the bottom should use the white pieces.

Pawns should fill both the second rank from the bottom and the top. The rooks should be placed in all the corners. The knights should be next to the rooks. The bishops should be next to the knights. Of the two squares remaining on the first rank, the queen should be placed on its colour (white on white or black on black) and the king on the other square.

https://www.chess.com/article/view/how-to-set-up-a-chessboard

How to play the game

ALL IMAGES AND INFORMATION FOR THIS SECTION FROM: chess.com

Once each player is ready, the light (white) colour always plays first. During each players turn, they can move one of their pieces to any possible square (within the movement constraints of the piece) that doesn't then cause that players king to be in check (page 20).

If a player moves its piece onto a square where their opponent's piece is, that piece is 'taken' and must be removed from the board.

Once a player finishes their turn, it becomes the opposing players turn. The alternating between the two players turns continues until the game ends.

Check:

When a king is being attacked, it is in check. This happens when a player moves one of its pieces in a position in which it could take its opponents king in that players next turn.

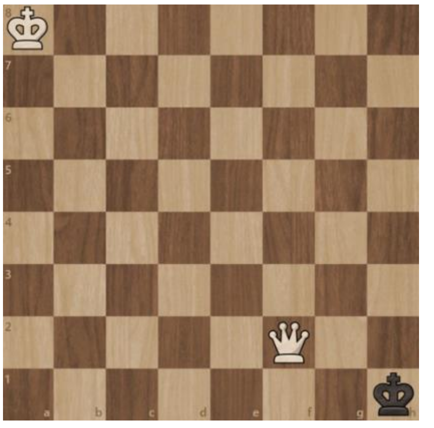
If a player's king is in check, they must move their king out of check in their next go.
This can be done by moving the king onto a square where it is no longer in range of any of the opponent's pieces, or by moving another piece in front of the king so it is blocking the attacking piece.
If a player's king is in check, and there is nothing they can do to move it out of check, this is called check mate and ends the game with the attacking player winning.

In the image opposite, black's king is in checkmate from the white knight since the king can't move out the way (due to no bordering spaces), and if the pawn takes the knight, then the king is still in check due to the white queen.

Stalemate:

Stalemate is a scenario in chess when there are no legal moves that can be made. If the king is not in check but no pieces can be moved without putting the king in check, this is a stalemate. A stalemate ends the game in a draw.

Stalemate can also be caused when the opponents king is their only piece left and they survive 50 moves.



Extra Chess Rules

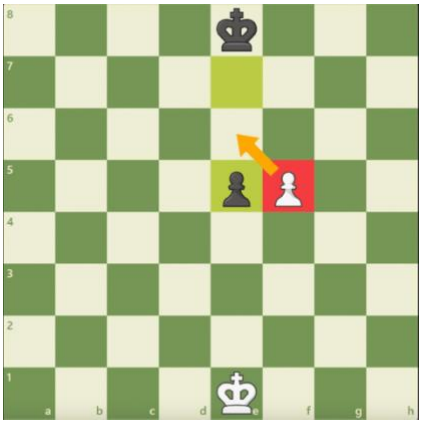
ALL IMAGES AND INFORMATION FOR THIS SECTION FROM: chess.com

En Passant:

En passant is a rule that allows a pawn to take another pawn without landing on its square.

There are a few rules for the move to be legal:

1. The taking pawn must have advanced at least three ranks from its position at the start of the game.
2. The taken pawn must have moved two squares in one move, landing right next to the taking pawn.
3. En Passant must be played immediately after the pawn being taken moves.



Castling:

Castling is the only time you can move two pieces in one turn.

There are 5 conditions for castling:

1. The king cannot have moved
2. The chosen rook cannot have moved
3. The king cannot be in check
4. The king cannot pass through check (during the move)
5. No pieces can be between the king and the rook

Once all these conditions are met, the king can be moved two spaces towards the chosen rook, and the rook will move to the other side of the king.

Chess Notation

All info for section from: https://www.chess.com/terms/chess-notation

Chess notation is a method of writing down all the moves in a chess game in a way that the game can be recreated just from the notation. The main method of chess notation is called algebraic notation.

Algebraic notation shows you the move number, the name of the piece that is moved and then the square where the piece moves. Each piece has its own abbreviation, and each square is named by its coordinates.

King: K
Queen: Q
Rook: R
Bishop: B
Knight: N
The pawn has no abbreviation. You know the pawn has moved if there is no abbreviation letter.

In the image opposite, the pawn moves two squares forward. Since it is the first move of the game, the notation starts with ‘1.’. The end square is ‘e4’. Therefore, the whole notation is ‘1.e4’.

In this example, the knight moves to f3. Since this is the third move of the game, and the night is abbreviated to ‘N’, the notation is: ‘3.Nf3’.

Special Cases:

For captures, castling, check, promoting and checkmate, the normal method of notation does not work. Some extra rules are added:

When a piece is captured an 'x' is then placed between the abbreviation and the coordinates. For example: 'Bxc6'. When a pawn captures a piece, you write the name of the file that the pawn is on, followed by 'x' and then the file where the pawn moves.

Castling on the king's side is recorded as '0-0' and castling on the queens side is recorded as '0-0-0'.

When the king is threatened 'check' you add a '+' to the end of the move. For example: 'Bb5+'. Similarly, checkmate is recorded with a # at the end of the move.

When a pawn promotes, a '=' is added to the end of the move followed by the abbreviation of the piece the pawn is promoting into.

Example:

This example chess game notation corresponds to the shortest game possible also known as the 'Fools mate':

e4 g5 Nc3 f5 Qh5#

The moves this notation represents are as follows:
- Move the pawn at e2 to e4
- Move the pawn at g7 to g5
- Move the knight at b1 to c3
- Move the pawn at f7 to f5
- Move the queen at d1 to h5
- The game ends in checkmate against black

Chess AI

Board + Piece Representation Methods

2D Array Board:

The board can be stored in a 2D array which represents a 8x8 matrix. This provides a simple, easily accessible method of storing the location of pieces and empty spaces. You would, however, have to check every board square to find the location of a specific piece.

It would look similar to this:

```
board = [
['b_rook','b_knight','b_bishop','b_queen','b_king','b_bishop','b_knight','b_rook']
['b_pawn','b_pawn','b_pawn','b_pawn','b_pawn','b_pawn','b_pawn','b_pawn'],
['e','e','e','e','e','e','e','e'],
['e','e','e','e','e','e','e','e'],
['e','e','e','e','e','e','e','e'],
['e','e','e','e','e','e','e','e'],
['w_pawn','w_pawn','w_pawn','w_pawn','w_pawn','w_pawn','w_pawn','w_pawn'],
['w_rook','w_knight','w_bishop','w_queen','w_king','w_bishop','w_knight','w_rook']
]
```

Where each piece is represented by the first letter of its colour followed by and underscore and then the name of the piece. An empty square is e.

https://cs.cornell.edu/boom/2004sp/ProjectArch/Chess/algorithms.html

Individual Piece Board:

Each piece could individually store its location and state. If there are no pieces with a location, then that location is considered empty. This would illuminate the need for a 2D array which would mean less data has to be stored. It would, however, be more complicated to navigate in the code and you would have to search all the pieces to check the state of a specific board square.

https://stackoverflow.com/questions/21374846/efficient-storage-of-a-chess-position
answer by ‘circular’

String oriented pieces:

Each piece could be represented by a string that identifies the type of piece and the colour. This could be the text in each square of the 2D array or the variable name for the individual piece. An empty square would have different identification.

Object oriented pieces

Each piece could be represented by an individual object that could be stored in its board square in the 2D array, or it could store its own board location as an attribute. The object would also be able to store states like weather En Passant or Castling is available.

https://cs.cornell.edu/boom/2004sp/ProjectArch/Chess/algorithms.html

Static Board Evaluation Function

When the min-max algorithm (page 25) gets down to the leaves of its search, it's unlikely that it reached a goal state (i.e. a checkmate). Therefore, it needs some way to determine whether the given board position is "good" or "bad" for it, and to what degree. A numerical answer is needed so that it can be compared to other board positions in a quantifiable way. Advanced chess playing programs can look at hundreds features of the board to evaluate it. The simplest, and perhaps most intuitive, look at only piece possession. Clearly, having a piece is better than not having one (in most cases at least).

Some additional features that could be included are:

- Pawn Advancement: How far up the board each pawn has reached. The further the better since the pawn is closer to being promoted into another piece.
- Piece mobility: How many different spaces can a piece move to.
- Piece threats: How many of the opponent's and AI's pieces are threatened by attack

https://www.cs.cornell.edu/boom/2004sp/ProjectArch/Chess/algorithms.html#staticboard

Min-max Algorithm

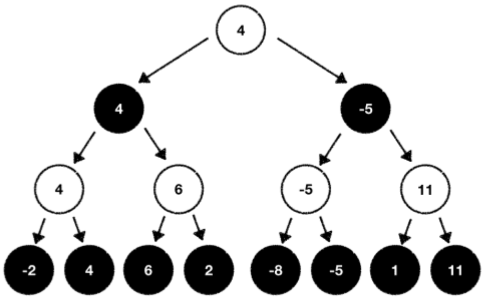
This algorithm will make up the core of the chess AI. It works by trying to MAXimise its own score, while MINimising its opponents score (relative to the AI):

When the AI starts its turn, all the possible AI moves, and their outcomes, are calculated. For each possible move, every one of the opponent's possible moves and outcomes are calculated. The computer's moves are then calculated again and the cycle goes on for as many times as a specified number which is called the 'depth'.

Each final outcome is then evaluated (using the static board evaluation function – page 25) and the move with the best outcome for the AI, given the player will play the best possible moves (to MINimise the computer score), is chosen. Therefore, the computer will MAXimise its score even if the player plays the best moves possible.

https://www.cs.cornell.edu/boom/2004sp/ProjectArch/Chess/algorithms.html#minmax

The image opposite is an example of a min-max tree, although it would be much bigger in a chess AI. The light colour nodes is the maximized score and the Dark colour nodes are the minimised score.
The Light colour nodes will pick the score (and board) with the largest score in all its leaves. For example, the highest node picks 4 out of 4 and -5. The Dark colour nodes will do the same but with the smallest score. For example the furthest top right dark node chooses -5 out of -5 and -11



https://philippmuens.com/minimax-and-mcts

This method can become very resource intensive and time consuming if the depth is too big since the number of outcomes to consider increases exponentially.

Alpha Beta Pruning

Alpha beta pruning is used to considerably decrease the min max search space. It keeps track of the worst and best moves for each player so far at each ply (ply = the depth the search is at) and using those you can avoid searching branches that are already guaranteed to yield worse results (you prune the branch). Using the alpha beta pruning will not result in the loss of any accuracy. Initially the alpha value should be negative infinity and the beta value should be positive infinity.

https://www.cs.cornell.edu/boom/2004sp/ProjectArch/Chess/algorithms.html#alphabeta

Quiescence Searching

Since the depth of the min-max search is limited, problems can occur at the frontier. A move that may seem great may actually be a disaster because of something that could happen on the very next move. Looking at all these possibilities would mean increasing the ply by 1, which is not the solution, as we would need to extend it to arbitrarily large depths. Most manoeuvres in chess result in only slight advantages or disadvantages to each player, not big ones at once. Hence, looking at higher depths is important only for significant moves - such as captures. Consider for example a move in which you capture the opponent's knight with your queen. If that is the limit of your min§ -max search, it seems to be a great move - you receive points for capturing the opponent's knight. But suppose that in the very next move your opponent can capture your queen. Then the move is clearly seen as bad, as trading a queen for a knight is to your disadvantage. Quiescence searching will be able to detect that by looking at the next move. Again, it doesn't need to do this for every move - just for ones that affect the score a lot (like captures).

https://www.cs.cornell.edu/boom/2004sp/ProjectArch/Chess/algorithms.html#quiescence

Opening Move Database

Many combinations of the first few initial moves on the board are known to establish a good position on the board. Thus, over the years, chess masters have developed what is essentially an encyclopaedia of these openings. The program can use databases like this
at the beginning of play - instead of performing the usual local search, it simply checks whether the moves performed so far fit into a given opening strategy, and then plays the appropriate response.

https://www.cs.cornell.edu/boom/2004sp/ProjectArch/Chess/algorithms.html#opening

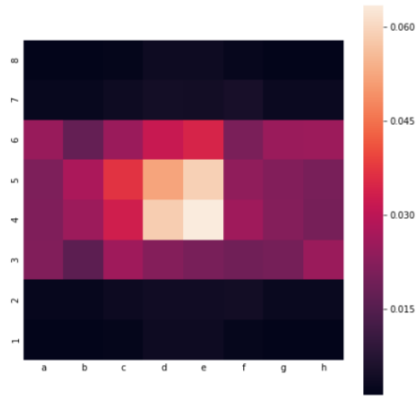
Heatmapping

A heatmap is a data visualisation technique that shows magnitude of a phenomenon as colour in two dimensions. The heatmap in chess will show which squares in a chess game are the most crucial ones. The AI can then use this map in its static board evaluation function to score the position of pieces on the board accordingly to the rating on the heatmap.

https://medium.com/nerd-for-tech/chess-heatmap-where-does-most-of-the-action-take-place-52b2a007dfa2#:~:text=A%20heat%20map%20(or%20heatmap,has%20been%20in%20the%20game.

The majority of good moves will be towards the middle of the board (rather than the edge) which means most heatmaps will show the middle of the board as the 'hottest'.

There are multiple examples of heatmaps on the internet and they can be created by analysing previously played games and where the majority of pieces were moved when the move was 'good'. An example is the image opposite where the higher numbers on the key mean better position.



https://www.reddit.com/r/dataisbeautiful/comments/eu4eok/heat_map_of_squares_played_on_a_chessboard_oc/

Requirements

1. Computer Based Player (CBP)
    1. Game must have a computer based player that plays in the turn of a player
        1. CBP must make its moves as per the rules of chess
        2. The CBP must make its move automatically when it's on its turn
    2. The CBP must have 3 levels of competency
        1. Make random moves
        2. Make the best move for that go (without looking ahead)
        3. Make the best move for that go (considering moves ahead)
    3. The CBP must make its move in reasonable time (1 min)

2. The game should have 3 difficulty settings
    1. The higher the difficulty setting, the harder the CBP should be to beat
        1. 0 = Easy
        2. 1 = Medium
        3. 2 = Hard
    2. Difficulty setting should be retrieved from the user at the start using a GUI window

3. The game should be displayed on a chess board
    1. Should be the correct 8x8 chess board structure with 64 squares
    2. The board should be labelled with letters on the x-axis and numbers on the y-axis according to the general chess board coordinate layout
    3. Should be obvious when a piece is in a square (and what square the piece is in)
    4. Each piece should be easy to identify according to the accepted general appearance of each chess piece
    5. The piece layout at the start of the game should be correct according to the rules of chess

4. A piece in the game should be movable using the displayed board
    1. A move should only be possible to make if it is legal according to the rules of chess
        1. Each piece should only be moved based on what the rules of chess dictate it can do
        2. A player cannot move itself into check
        3. If a move ends on an enemy piece, the enemy piece should be deleted (taken)
    2. Special moves such as En Passant, Castling and Promotion should all be possible and easy to use according to their specific move rules
        1. En Passant can only occur when:
            1. The taking pawn has advanced at least 3 ranks
            2. The taken pawn has moved two squares in one move

        3. The taken pawn is right next to the taking pawn
        4. The move is played immediately after the pawn being taken has moved
    2. Castling can only occur when:
        1. The space between the chosen rook and the king is all empty
        2. Neither the rook nor the king have moved yet
        3. The king is not moving out of check
    3. Promotion can only occur when:
        1. The pawn being promoted is on the other side of the board
        2. When a pawn is on the other side of the board IT MUST BE PRPOMOTED
3. When a piece is moved, the new board state should be shown on the displayed board

5. Special scenarios should be automatically detected and dealt with
    1. Check for Check
        1. If a king is threatened by an opponent's piece, it is in check
    2. Check for Checkmate
        1. If the king is in check and there are no possible moves for the player to make, it is in checkmate
        2. If a king is in checkmate, the user should be notified, and the game should end
    3. Check for stalemate
        1. If a king is not threatened by an opponent's piece, but there is no possible move to make, this is a stalemate
        2. The user should be notified, and the game should end (in a draw)

6. There must be an option to continue the previous game whenever starting the program
    1. The game must be automatically saved after every move
        1. The board state must be saved using a specific format for it to be read again by the program
        2. Whoever's turn it is next must be saved
    2. A GUI must be used to ask (and get a response from) the user if they want to load the previous game or start fresh

Requirement Justification

1. In the interview and the questionnaire, it was concluded that people see a computer based chess game as a good solution to the two-player issue.
2. In the questionnaire it was concluded that most people preferred to have only a few difficulty modes.
3. As concluded in the interview the game should be clear and easy to understand. Board layout and structure is on pages 14 and 19.
4. In the interview, it was concluded that pieces should be moved using the GUI. The rules of chess that govern the moves are on pages 15 to 21. Chess notation is on page 22 which is used for the move logging.
5. Check, checkmate, and stalemate are on pages 20 and 21.
6. In the interview it was concluded that it would be useful to have the board save and load feature.

Design

Class Structure

The system will use 7 different classes:

UserInterface – Will manage the game user interface. Will include the code from the Graphical User Interface section of the design.

Game – Will manage the game with a loop that will continue until there is a game ending scenario/event. The game will be responsible for getting moves and passing them through to the board to validate and amend the board array.

Board – Will store the board array along with having multiple methods to amend make moves, validate moves, check for scenarios (check etc) and get valid moves

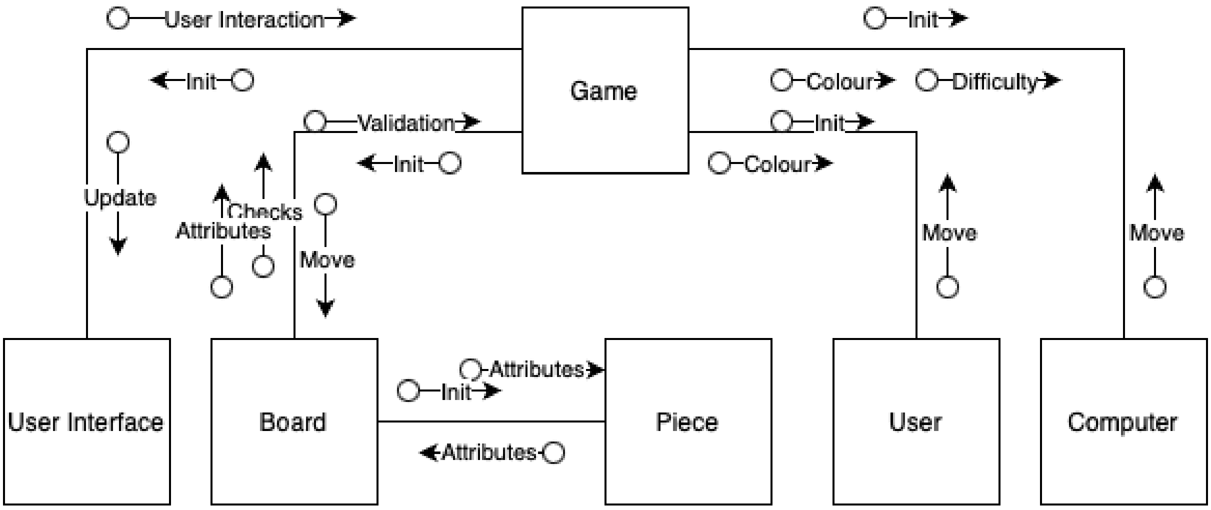
Piece – Will represent each individual piece in the game. Will store the piece type, colour, and position along with weather the piece has been moved yet or if it is susceptible to en passant. It will also contain multiple access methods to change and get the values from the attributes

Player – A parent class the contains the basic attributes and methods for a player in the game. The player class stores the player colour and has a method that gets the colour.

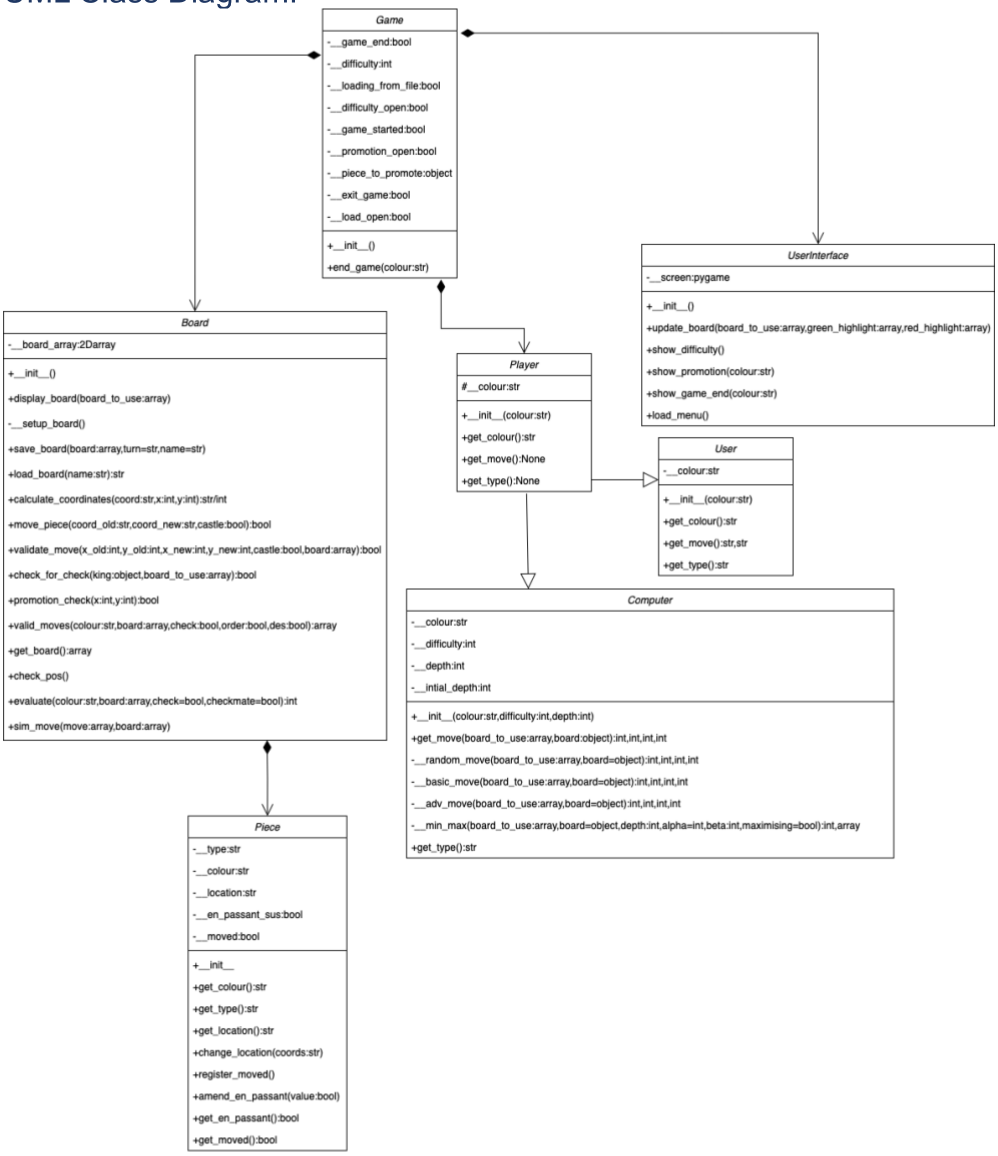
User – A child class of 'Player' which represents a human player that is using the chess game software. The User class has a method to get the move from the player using input statements to input coordinates of the piece to be moved and the place to move it too.

Computer – A child class of 'Player' which represents a computer player (AI). The computer class has a method to get the move from the computer using a minmax algorithm.

Structure Chart



UML Class Diagram:



Required Packages

Python library ‘copy’ is required to deep copy an array.
Python library ‘abc’ is required to make the ‘Player’ class abstract
Python library ‘pygame’ is required to make the GUI

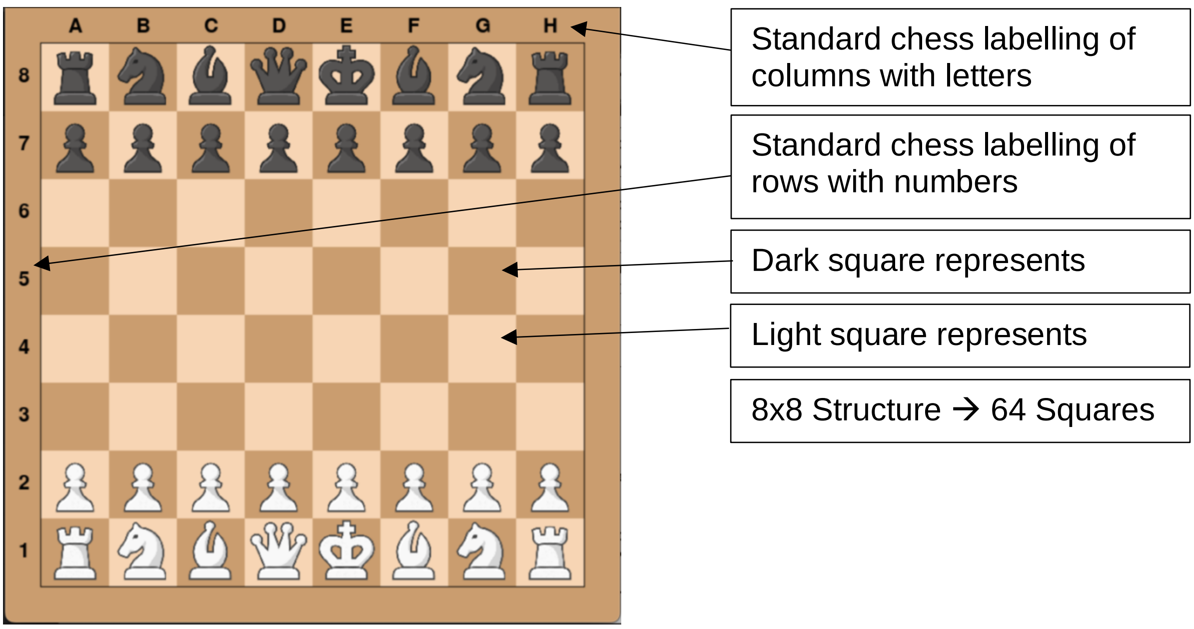
Graphical User Interface

Chess Board

Board

The most interacted with GUI will be the chess board. This should be interactive and easy to understand. The board should also be labelled with the letter and number coordinates on the columns and rows respectively.

- Image from https://www.vectorstock.com/royalty-free-vector/chess-board-without-pieces-vector-1765482



The chess board GUI will simply be a graphical representation of the chess board array which will be updated whenever there are any changes.

When a player clicks on a board square, it is detected what square the player has clicked on. This board square represents the location of the piece that is to be moved. The player then clicks on the board square they want the piece to move to. If the move is legal, the board will be updated and the computer will make its move, and if not, the whole turn will be reset and will start from the beginning.

Pieces

Each piece on the chess board needs to be represented by an easily understandable image that resembles the standard chess piece symbol. The colours should also be distinguishable (black and white).

Here are the images for the pieces:

Piece	**White Image**	**Black Image**	**Reference**
Pawn			https://chess.fandom.com/wiki/Pawn
Rook			https://chess.fandom.com/wiki/Rook
Knight			https://chess.fandom.com/wiki/Knight
Bishop			https://chess.fandom.com/wiki/Bishop
Queen			https://chess.fandom.com/wiki/Queen
King			https://chess.fandom.com/wiki/King

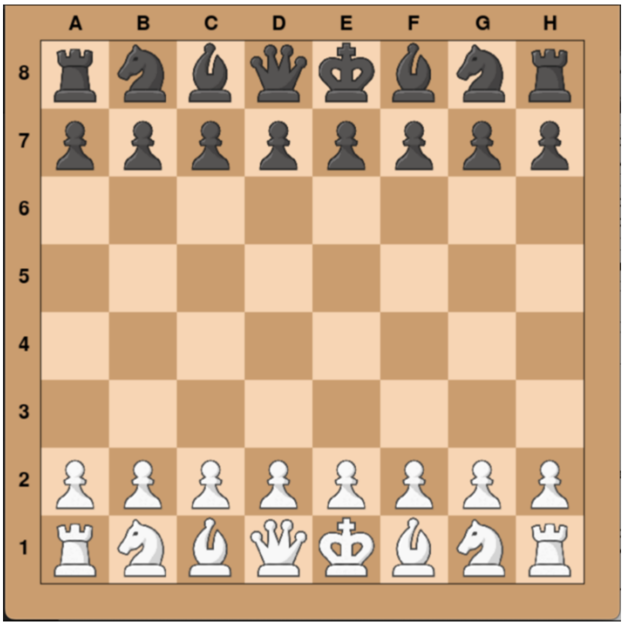
The pieces will be organised at the start of the game according to the rules of chess as following (using chess coordinate notation):

White Pawns from A2 – H2
White Rooks at A1 and H1

White Knights at B1 and G1
White Bishops at C1 and F1
White Queen at D1
White King at E1

Black Pawns from A7 – H7
Black Rooks at A8 and H8
Black Knights at B8 and G8
Black Bishops at C8 and F8
Black Queen at D8
Black King at E8

Or as shown on this image:



This image also shows what it looks like when a piece is in a square. It does not cross any lines to make it clear which square the piece is in.

The window will be initialised using the pygame syntax and the board will be created using the same code that the Board Updating does.

Board Updating

The board will be updated using the 'update_board' method every time a change is made. The update board method also takes two arrays: green_highlight and red_highlight which contain a list of coordinates of squares that need to be highlighted in either green or red respectively.

Pseudocode:

```
SUBROUTINE update_board(board_to_use, green_highlight, red_highlight)
```

```
CONSTANT LIGHT_COLOUR ← (246, 213, 180)
CONSTANT DARK_COLOUR ← (202, 157, 111)
CONSTANT EDGE_COLOUR ← (0, 0, 0)

margin ← CREATE_SURFACE((680, 680))
FILL_SURFACE(margin, DARK_COLOUR)
BLIT_SURFACE(self.__screen, margin, (0, 0))

margin2 ← CREATE_SURFACE((602, 602))
FILL_SURFACE(margin2, EDGE_COLOUR)
BLIT_SURFACE(self.__screen, margin2, (39, 39))

font ← CREATE_FONT('arialblack.ttf', 32)
FOR i FROM 0 TO 7
   letter ← RENDER_TEXT(chr(i + 65), True, EDGE_COLOUR, font)
   number ← RENDER_TEXT(STR(8 - i), True, EDGE_COLOUR, font)
   BLIT_SURFACE(self.__screen, letter, (i * 75 + 70, 10))
   BLIT_SURFACE(self.__screen, number, (15, i * 75 + 65))
END FOR

board ← CREATE_SURFACE((600, 600))

FOR x ← 0 TO 7 STEP 2
   FOR y ← 0 TO 7 STEP 2
      DRAW_RECTANGLE(board, LIGHT_COLOUR, (x * 75, y * 75, 75, 75))
   END FOR
END FOR

FOR x ← 1 TO 7 STEP 2
   FOR y ← 1 TO 7 STEP 2
      DRAW_RECTANGLE(board, LIGHT_COLOUR, (x * 75, y * 75, 75, 75))
   END FOR
END FOR

FOR x ← 1 TO 7 STEP 2
   FOR y ← 0 TO 7 STEP 2
      DRAW_RECTANGLE(board, DARK_COLOUR, (x * 75, y * 75, 75, 75))
   END FOR
END FOR

FOR x ← 0 TO 7 STEP 2
   FOR y ← 1 TO 7 STEP 2
      DRAW_RECTANGLE(board, DARK_COLOUR, (x * 75, y * 75, 75, 75))
   END FOR
END FOR

BLIT_SURFACE(self.__screen, board, (40, 40))
```

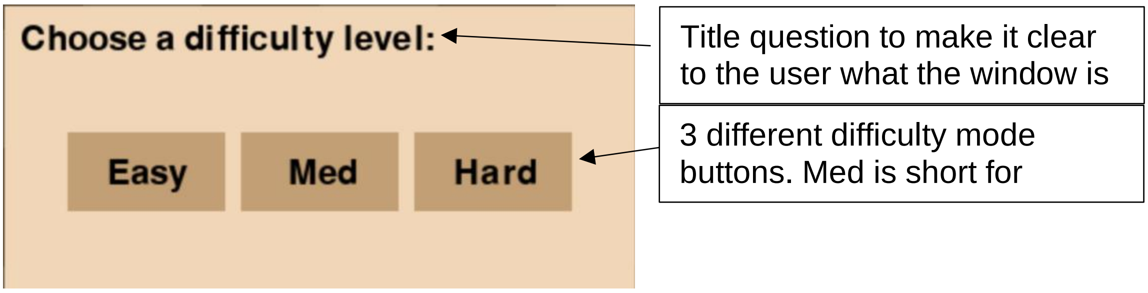
```
    FOR square IN green_highlight
        x ← square[0]
        y ← square[1]
        green_square ← CREATE_SURFACE((75, 75), SRCALPHA, 32)
        FILL_SURFACE(green_square, (0, 200, 0, 50))
        BLIT_SURFACE(self.__screen, green_square, (75 * x + 40, 75 * y + 40))
    END FOR

    FOR square IN red_highlight
        x ← square[0]
        y ← square[1]
        red_square ← CREATE_SURFACE((75, 75), SRCALPHA, 32)
        FILL_SURFACE(red_square, (255, 0, 0, 50))
        BLIT_SURFACE(self.__screen, red_square, (75 * x + 40, 75 * y + 40))
    END FOR

    FOR row ← 0 TO 7
        FOR column ← 0 TO 7
            square ← board_to_use[row][column]
            IF square ≠ NULL
                t ← LOWERCASE(square.get_type())
                c ← LOWERCASE(square.get_colour())
                image_obj ← LOAD_IMAGE(f'pieces/{t}_{c}.webp')
                image_obj ← SCALE_IMAGE(image_obj, (80, 80))
                BLIT_SURFACE(self.__screen, image_obj, (75 * column + 37, 75 * row
+ 35))
            END IF
        END FOR
    END FOR

    UPDATE_DISPLAY()

END SUBROUTINE
```

This uses predicted GUI locations

Difficulty Selection

There will be 3 difficulty modes: easy, medium, and hard. This will determine how difficult the AI is to beat.

The difficulty setting will be retrieved from the player at the start of the game using the difficulty selection window:



The button presses are detected by getting the location of each mouse click. The difficulty is than stored in a variable in the Game object.

Pseudocode for getting mouse click:

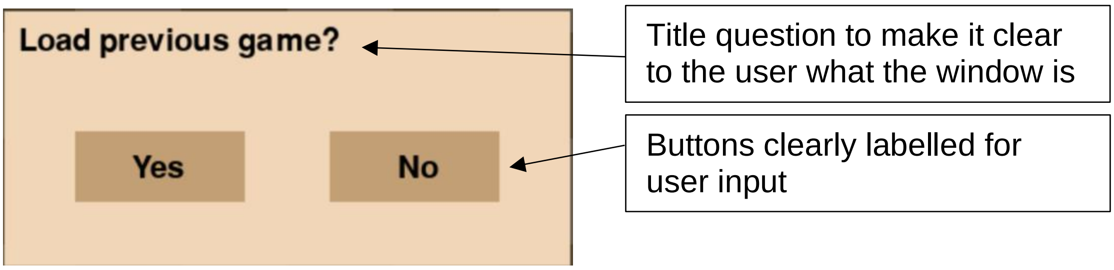
```
IF pygame.mouse.get_pressed()[0] AND NOT pressed THEN
   pressed ← True
   x_pos ← pygame.mouse.get_pos()[0]
   y_pos ← pygame.mouse.get_pos()[1]

   IF x_pos > 180 AND x_pos < 280 AND y_pos > 330 AND y_pos < 380 THEN
      self.__difficulty ← 0
      UI.update_board(board.get_board())
      difficulty_open ← False
   ELSE IF x_pos > 290 AND x_pos < 390 AND y_pos > 330 AND y_pos < 380
THEN
      self.__difficulty ← 1
      UI.update_board(board.get_board())
      difficulty_open ← False
   ELSE IF x_pos > 400 AND x_pos < 500 AND y_pos > 330 AND y_pos < 380
THEN
      self.__difficulty ← 2
      UI.update_board(board.get_board())
      difficulty_open ← False
   ENDIF
ENDIF
```

This uses predicted coordinates of the edge of the buttons

Board Loading

There will be an option to load a board from last game or start fresh which will be displayed to the user before the game starts.

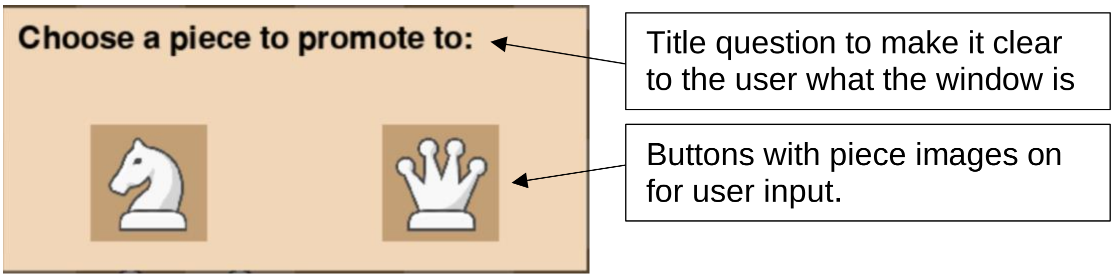
The board load screen will look like this:



The button presses are detected by getting the location of each mouse click in the main program in a similar way to the get difficulty screen.

Piece Promotion

Whenever a player's pawn gets to the other side of the board it is promoted. This is what the piece promotion screen looks like:



The button presses are detected by getting the location of each mouse click in the main program in a similar way to the get difficulty screen.

End Game Screen

When a game ends, a message is displayed in the middle of the screen stating the conditions of the game ending. It looks like this:

Data Structures

Board

The board will be stored in a 2D array. The outer part of the array will be a list of rows and the inner part a list of squares (locations).
Each location in the array will either contain a ‘None’ type (which will indicate the array is empty) or an object which will represent a piece.

The array be initialised with the starting positions of a chess board and will look like this:

```
__board_array ← [
   [Piece('R','a8', 'B'), Piece('N','b8', 'B'), Piece('B','c8', 'B'), Piece('Q','d8', 'B'),
Piece('K','e8', 'B'), Piece('B','f8', 'B'), Piece('N','g8', 'B'), Piece('R','h8', 'B')],
   [Piece('P','a7', 'B'), Piece('P','b7', 'B'), Piece('P','c7', 'B'), Piece('P','d7', 'B'),
Piece('P','e7', 'B'), Piece('P','f7', 'B'), Piece('P','g7', 'B'), Piece('P','h7', 'B')],
   [None, None, None, None, None, None, None, None],
   [None, None, None, None, None, None, None, None],
   [None, None, None, None, None, None, None, None],
   [None, None, None, None, None, None, None, None],
   [Piece('P','a2', 'W'), Piece('P','b2', 'W'), Piece('P','c2', 'W'), Piece('P','d2', 'W'),
Piece('P','e2', 'W'), Piece('P','f2', 'W'), Piece('P','g2', 'W'), Piece('P','h2', 'W')],
   [Piece('R','a1', 'W'), Piece('N','b1', 'W'), Piece('B','c1', 'W'), Piece('Q','d1', 'W'),
Piece('K','e1', 'W'), Piece('B','f1', 'W'), Piece('N','g1', 'W'), Piece('R','h1', 'W')]
]
```

This is done inside a __setup_board() method

Each piece is initialised using its type, colour, and location.

When trying to access a location on the board, the index works as follows: ‘board_array[row_index][column_index]’

Due to these indexes not being the same as chess notation for coordinates, a calculate coordinates method is used to convert between the two. Here is the pseudocode for that method:

```
SUBROUTINE calculate_coordinates(coord, x, y)
   IF coord NOT EQUAL TO None THEN
      x ← GET_FIRST_ELEMENT(coord)
      y ← CONVERT_TO_INT(GET_SECOND_ELEMENT(coord))
      RETURN (ASCII_VALUE(x) - 97), 7 - (y - 1)
   ELSE IF x NOT EQUAL TO None AND y NOT EQUAL TO None THEN
      RETURN CONCATENATE(CHARACTER_FROM_ASCII(x + 97), 7 - (y - 1))
   ENDIF
ENDSUBROUTINE
```

As you can see in this pseudocode, the method will return the index x and y if a value for coord is passed, or a chess coordinate if values for x and y are passed.

Board Saving

When the board is saved, it needs to be done so in a way that can be read again. Therefore, it needs to be done using a ‘standard’.

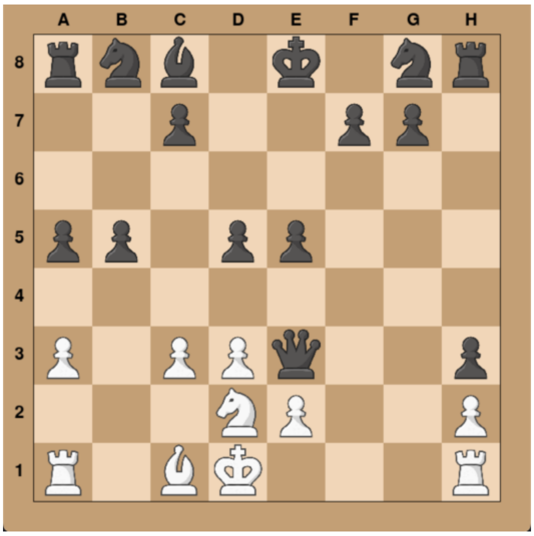
For the chess game, I have created my own standard that works as follows:

- The start of the saved file will be a letter, wither W or B, which indicates what colour has their turn next
- The rest of the saved file consists of 64 squares that are represented individually using square brackets:
    - If the square brackets contain ‘None’ e.g: ‘[None]’, then the board square is empty
    - For all other board squares with pieces on, the brackets will contain 3 pieces of information each separated by 1 space: piece type, piece location, piece colour. E.g: ‘[N b8 B]’ is a black knight at space B8
    - Each square is also separated by one space
- While the squares are saved in order according to how the pieces are displayed on the board to make it easier to read and write, they can actually be in any order and the only thing that matters in the placement of a piece on the board is its written location. This also means there doesn’t have to be all 64 squares and only the pieces have to be listed.

Here is an example of a file:

```
W [R a8 B] [N b8 B] [B c8 B] [None] [K e8 B] [None] [N g8 B] [R h8 B] [None]
[None] [P c7 B] [None] [None] [P f7 B] [P g7 B] [None] [None] [None] [None]
[None] [None] [None] [None] [None] [P a5 B] [P b5 B] [None] [P d5 B] [P e5 B]
[None] [None] [None] [None] [None] [None] [None] [None] [None] [None] [None] [P
a3 W] [None] [P c3 W] [P d3 W] [Q e3 B] [None] [None] [P h3 B] [None] [None]
[None] [N d2 W] [P e2 W] [None] [None] [P h2 W] [R a1 W] [None] [B c1 W] [K d1
W] [None] [None] [None] [R h1 W]
```

This translates to the following board:



The board is saved using the save_board method which takes 3 parameters:

- board = the board that needs to be saved (2D array)
- turn = what colour has their turn next (string)
- name = file name (string)

Pseudocode for save_board:

```
SUBROUTINE save_board(board, turn, name)
    IF board = None THEN
        board ← self.__board_array
    ENDIF

    final_save ← turn + ' '

    FOR row IN board
        FOR square IN row
            IF square = None THEN
                final_save ← final_save + '[None] '
            ELSE
                final_save ← final_save + [type + location + colour]

```

The saved board is read and loaded into the game using the load_board method which takes 1 parameter:

- name = file name (string)

This method also returns the colour of the player who’s turn it is next

Pseudocode for load_board:

```
SUBROUTINE load_board(self, name='board.txt')
    file ← OPEN(name, 'r')
    text ← READ(file)
    CLOSE(file)

    new_board ← [
        [None, None, None, None, None, None, None, None],
        [None, None, None, None, None, None, None, None],
        [None, None, None, None, None, None, None, None],
        [None, None, None, None, None, None, None, None],
        [None, None, None, None, None, None, None, None],
        [None, None, None, None, None, None, None, None],
        [None, None, None, None, None, None, None, None],
        [None, None, None, None, None, None, None, None]
    ]

    turn ← SUBSTRING(text, 0, POSITION(text, ' '))
    text ← SUBSTRING(text, POSITION(text, ' ') + 1)

    WHILE text ≠ ‘’ DO
```

```
        object_str ← SUBSTRING(text, POSITION(text, '[') + 1, POSITION(text, ']'))
        text ← SUBSTRING(text, POSITION(text, ']') + 2)

        IF object_str = 'None' THEN
            CONTINUE
        ELSE
            piece_type ← SUBSTRING(object_str, 0, POSITION(object_str, ' '))
            object_str ← SUBSTRING(object_str, POSITION(object_str, ' ') + 1)
            piece_loc ← SUBSTRING(object_str, 0, POSITION(object_str, ' '))
            object_str ← SUBSTRING(object_str, POSITION(object_str, ' ') + 1)
            piece_colour ← object_str

            x, y ← calculate_coordinates(coord=piece_loc)
            new_board[y][x] ← Piece(piece_type, piece_loc, piece_colour)
        ENDIF
    ENDWHILE

    self.__board_array ← COPY(DEEPCOPY(new_board))
    RETURN turn
ENDSUBROUTINE
```

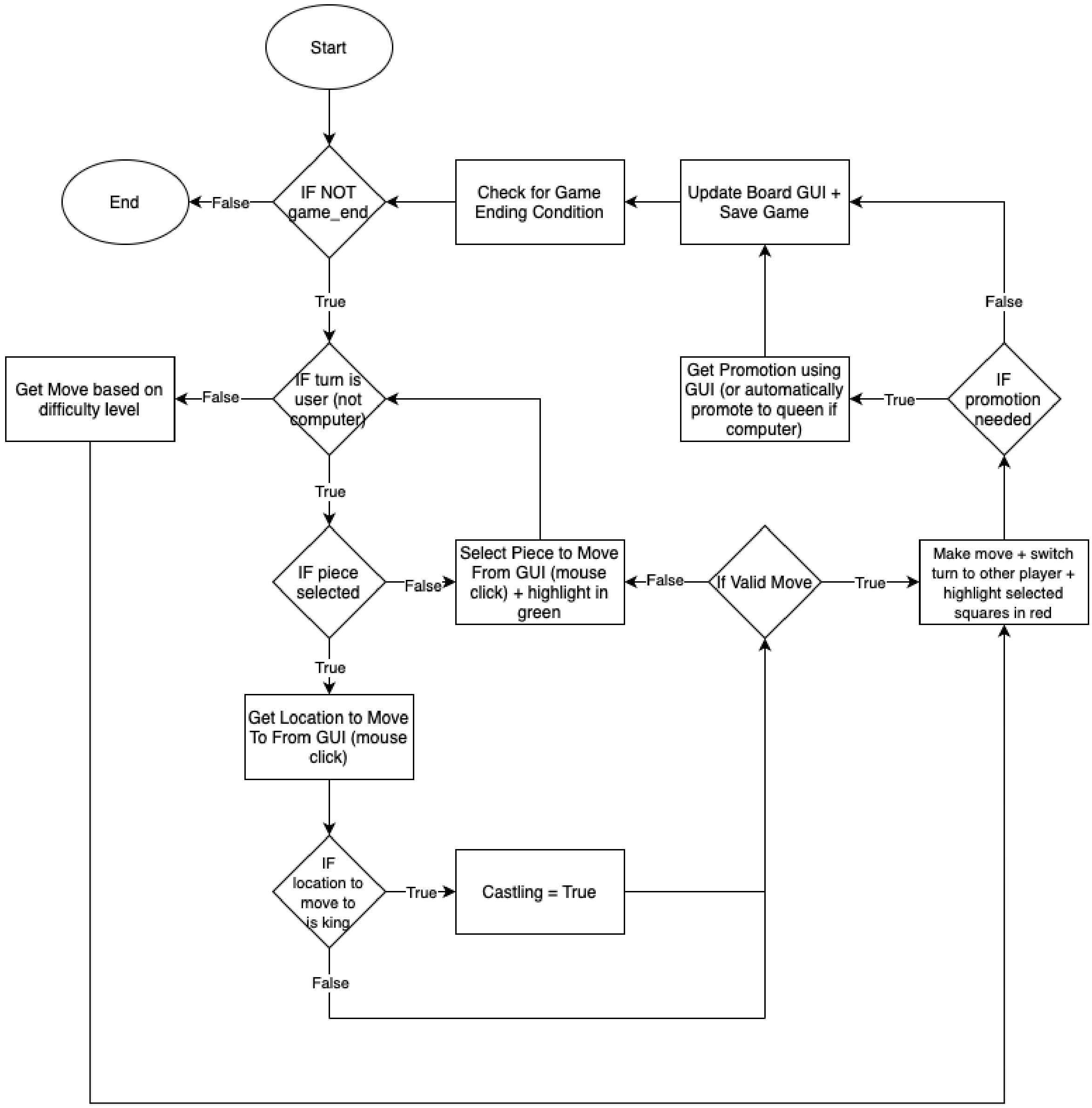
Major Processes

Main Loop

After retrieving all essential info from the user, like difficulty and board loading, the program will enter a main loop that will continue until it detects a game ending condition (checkmate or stalemate).

The main loop will detect who's turn it is and carry that turn out as a computer or user move based on what type the player is. The main loop will also handle detection for castling and will control the GUI for promotions and game ending messages.

Flowchart:



At the end of each move, the game will also be automatically saved to a file.

Move Piece

This function is used to edit the board to move a piece. It returns a boolean value of True if it was successful or false if not. The move piece function uses the validate move function to check the move it is making is valid before making it.

The function will take 3 parameters:

- coord_old = the start coordinates of the piece to be moved in chess coordinate form (string)
- coord_new = the end coordinates of the piece to be moved in chess coordinate form (string)
- castle = true when the move involves castling (defaults as false) (boolean)

Pseudocode:

```
# Calculate coordinates of the old position
x_old, y_old ← self.calculate_coordinates(coord_old)

# If not castling, calculate coordinates of the new position
IF NOT castle THEN
   x_new, y_new ← self.calculate_coordinates(coord_new)
ELSE:
   x_new ← None
   y_new ← None
ENDIF

# Retrieve piece type and colour from the old position
piece_type ← self.__board_array[y_old][x_old].get_type()
piece_colour ← self.__board_array[y_old][x_old].get_colour()

# Validate the move
validation ← self.validate_move(x_old, y_old, x_new, y_new, castle)

# If validation successful and castling
IF validation AND castle THEN
   # Perform castling
   IF x_old EQUALS 0 THEN
      # Queen-side castling
      self.__board_array[y_old][3] ← self.__board_array[y_old][x_old]
      self.__board_array[y_old][3].change_location("d" + ABS(y_old - 8))
      self.__board_array[y_old][x_old] ← None
      self.__board_array[y_old][2] ← self.__board_array[y_old][4]
      self.__board_array[y_old][2].change_location("c" + ABS(y_old - 8))
      self.__board_array[y_old][4] ← None

      self.__board_array[y_old][3].register_moved()
      self.__board_array[y_old][2].register_moved()
   ELSE IF x_old EQUALS 7 THEN
      # King-side castling
```

```
        self.__board_array[y_old][5] ← self.__board_array[y_old][x_old]
        self.__board_array[y_old][5].change_location("f" + ABS(y_old - 8))
        self.__board_array[y_old][x_old] ← None
        self.__board_array[y_old][6] ← self.__board_array[y_old][4]
        self.__board_array[y_old][6].change_location("g" + ABS(y_old - 8))
        self.__board_array[y_old][4] ← None

        self.__board_array[y_old][5].register_moved()
        self.__board_array[y_old][6].register_moved()

    ENDIF

    # Reset en passant for all pieces
    FOR EACH row IN self.__board_array DO
        FOR EACH piece IN row DO
            IF piece NOT EQUALS None THEN
                piece.amend_en_passant(False)
            ENDIF
        END LOOP
    END LOOP
    RETURN True

# If validation successful and not castling
ELSE IF validation THEN
    # Move the piece to the new position
    self.__board_array[y_new][x_new] ← self.__board_array[y_old][x_old]
    self.__board_array[y_new][x_new].change_location(coord_new)
    self.__board_array[y_old][x_old] ← None

    self.__board_array[y_new][x_new].register_moved()

    # Reset en passant for all pieces
    FOR EACH row IN self.__board_array DO
        FOR EACH piece IN row DO
            IF piece NOT EQUALS None THEN
                piece.amend_en_passant(False)
            END IF
        END FOR
    END FOR

    # If it's a pawn move of two squares, update en passant status
    IF piece_type EQUALS 'P' AND ABS(y_new - y_old) EQUALS 2 THEN
        self.__board_array[y_new][x_new].amend_en_passant(True)
    END IF

    RETURN True

# If validation successful and castling and pawn promotion
```

```
ELSE IF validation EQUALS (True, True) THEN
   # Move the piece to the new position and clear the old position and promotion
square
   self.__board_array[y_new][x_new] ← self.__board_array[y_old][x_old]
   self.__board_array[y_new][x_new].change_location(coord_new)
   self.__board_array[y_old][x_old] ← None
   self.__board_array[y_old][x_new] ← None

   self.__board_array[y_new][x_new].register_moved()

   # Reset en passant for all pieces
   FOR EACH row IN self.__board_array DO
      FOR EACH piece IN row DO
         IF piece NOT EQUALS None THEN
            piece.amend_en_passant(False)
         END IF
      END FOR
   END FOR

   RETURN True

# If validation failed
ELSE:
   RETURN False
END IF
```

Validate Move

The validate move function is used inside the move piece function and returns a boolean value (or a tuple of 2 boolean values) based on weather a passed move is valid or not and what the type of move is.

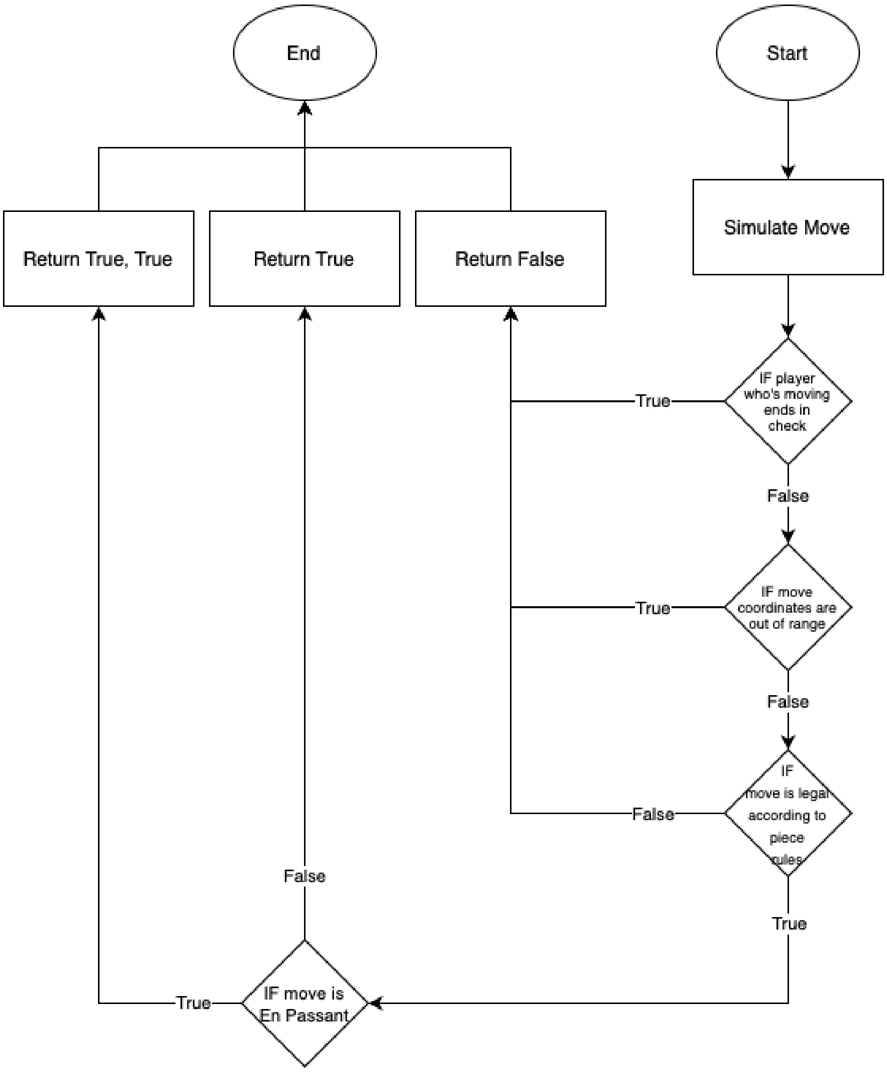
If 'True' is returned, the move is valid and is considered a 'normal' or 'castling' move. If 'False' is returned, then the move is considered invalid no matter the type. If 'True, True' is returned, then the move is considered a valid En Passant move.

The function works by first checking that the player is not putting themselves into check, followed by checking the coordinates are in the acceptable range and then checking that the piece that is being moved follows its unique chess rules. This includes for castling and En Passant.

The function will take 6 parameters:

- x_old = the start x location of the move to be validated in index form (integer)
- y_old = the start y location of the move to be validated in index form (integer)
- x_new = the end x location of the move to be validated in index form (integer)
- y_new = the end y location of the move to be validated in index form (integer)
- castle = true when the move involves castling (defaults as false) (boolean)
- board = the board the validation should be done on (2d array)

Flowchart:



Check For Check

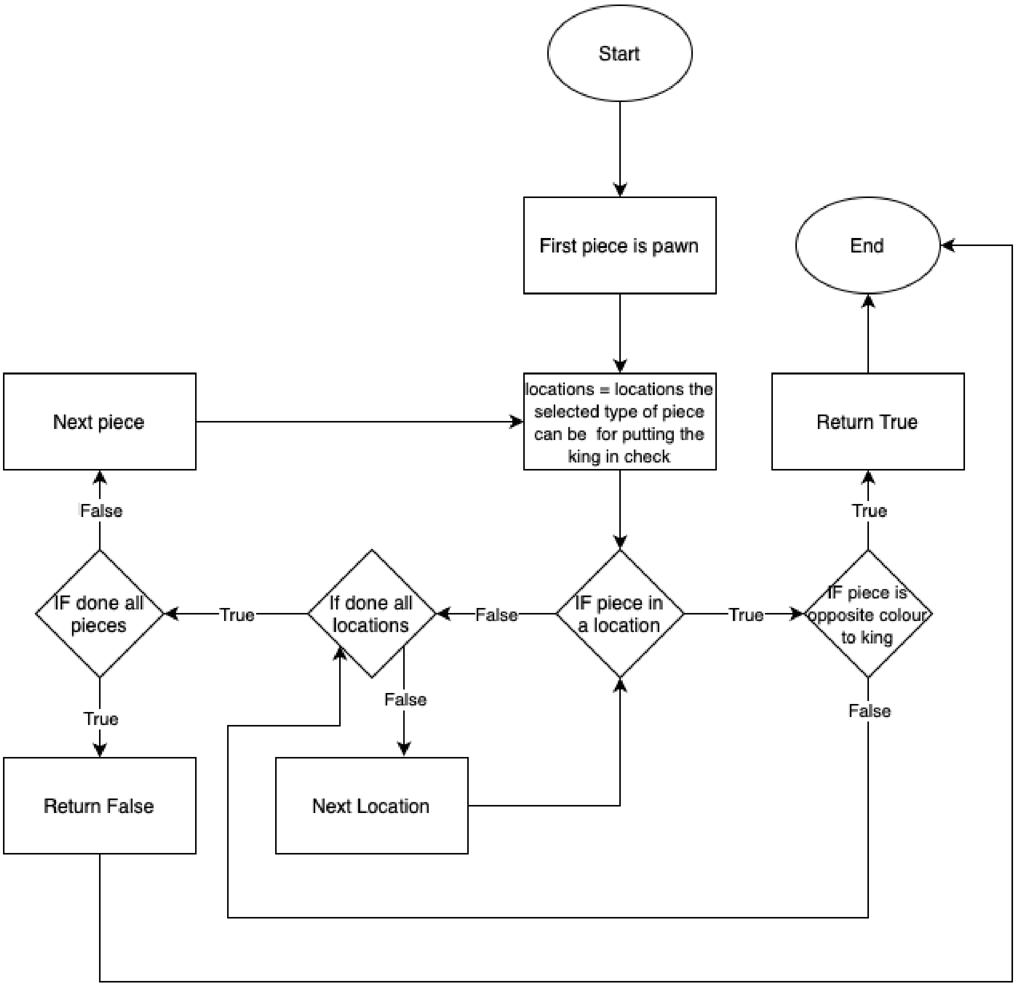
The check for check function takes a board and a king piece as parameters. The function than uses the board to check locations around the king for pieces that are putting it in check. It does this for every type of piece at every square they could be in for putting the king in check.

A value of 'True' is returned if the king is in check, or 'False' if not.

The function takes 2 parameters:

- king = the king the check is being carried out on (object)
- board_to_use = the board that the check should be carried out on (2d array)

Flowchart:



Check for Checkmate + stalemate

Checking for checkmate and stalemate is done at the end of every move and after pawn promotions.

It works by checking if there is a check when there are no possible moves to make. If there is a check, the game ends with checkmate, if not then the game will end with stalemate.

This code is not its own function.

Pseudocode:

```
IF valid_moves = [] THEN
    IF king_in_check THEN
        end_game checkmate
    ELSE
        end_game stalemate
    END IF
END IF
```

Get Valid Moves

This function will take a 4 parameters:

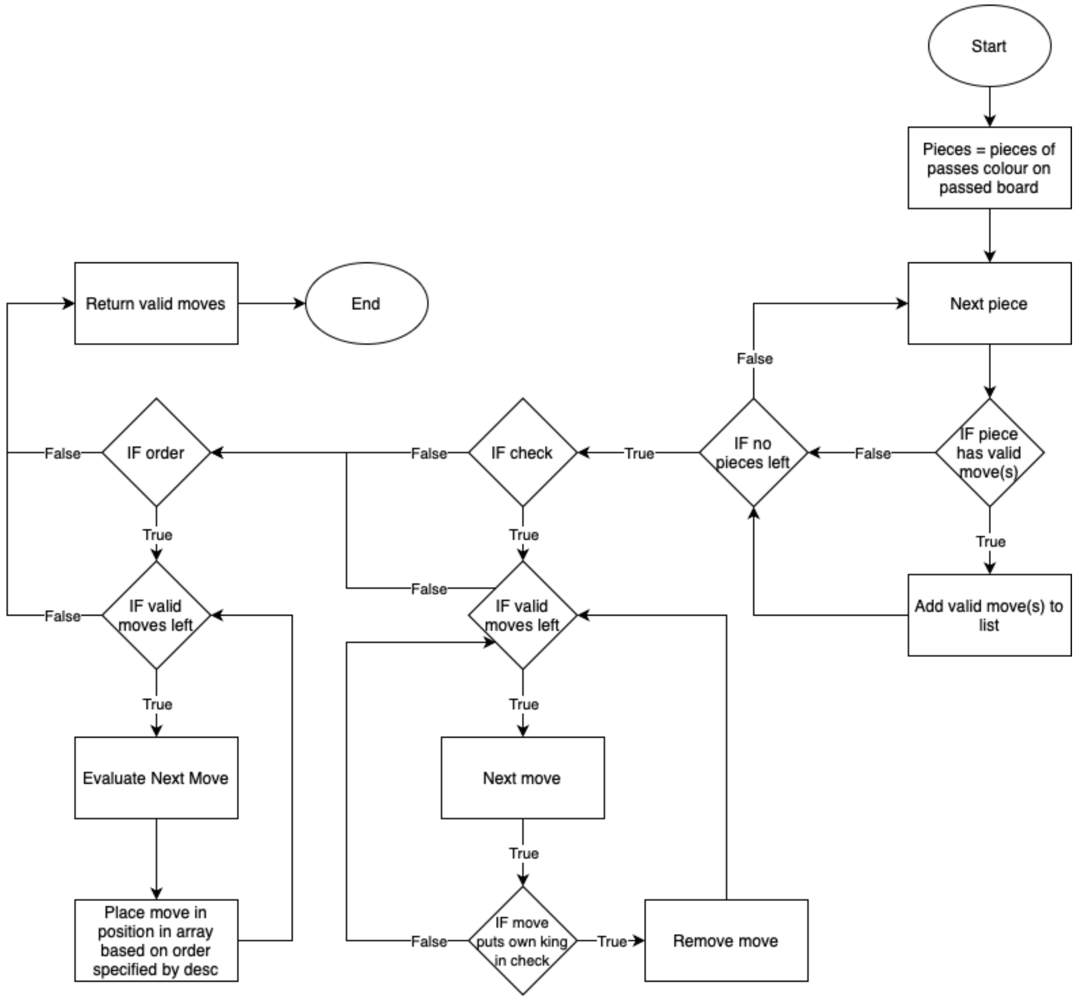
- colour = the colour that the moves will be generated for (string)
- board = the board to generate valid moves on (2d array)
- check = whether the moves should account for check rules (boolean)
- order = whether the moves should be automatically ordered based on their heuristic score after they have been simulated (boolean)
- des = the order in which the moves will be placed (if the order parameter is true) (boolean)

The valid moves are calculated by running through all the pieces of the passed colour. For each piece, the possible locations current piece would be allowed to move to are individually checked that they valid places for the piece to move to with the board state (according to chess rules). If they are, the move is added to the valid moves array that is returned from the function at the end.

If the function is checking for check, each valid move that has been collected will have their end board state check that they are not moving into check. If they are, the move will be purged.

If the function is ordering, each move outcome is evaluated, and it is placed back into its list in order of evaluation score.

Flowchart:



Evaluation

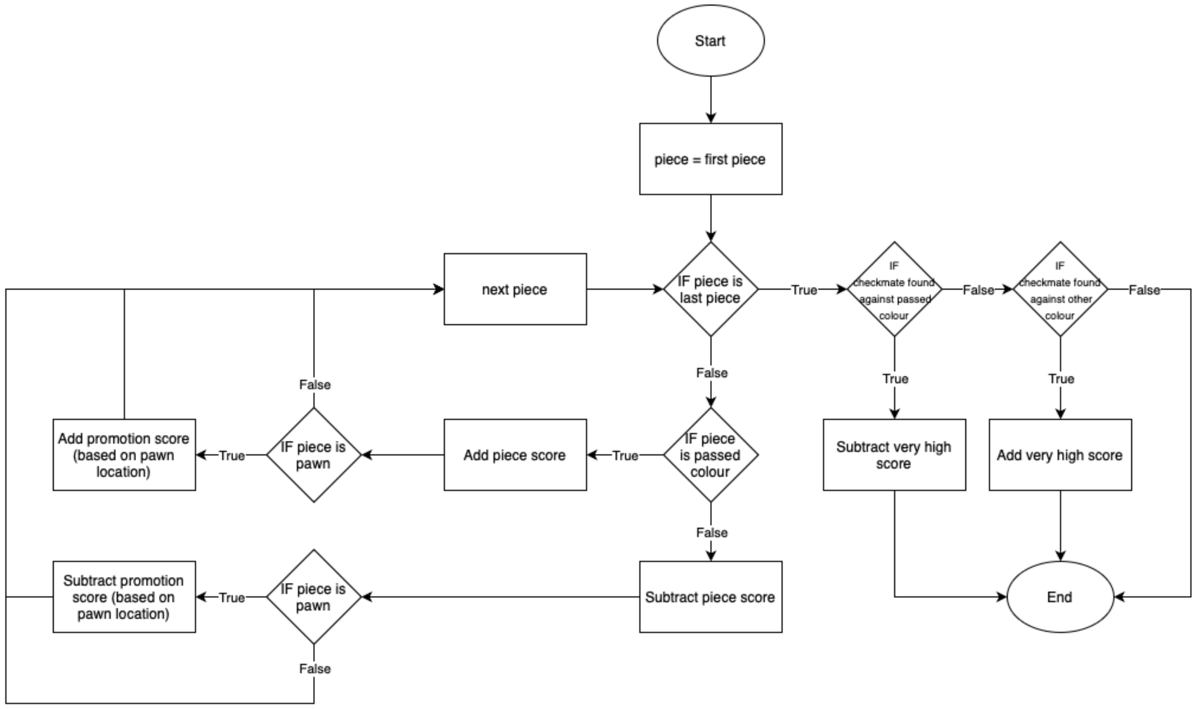
This function takes 4 parameters:

- colour = the colour the board will be evaluated relative too (string)
- board = the board to be evaluated (array)
- check = whether or not checks will be considered (bool)
- checkmate = whether or not checkmates will be considered (bool)

The board is evaluated based on multiple criteria:

1. Piece possession: each type of piece is scored based on their value (pawn lowest, queen highest). The total score for each sides piece possession is added to the evaluation score (or subtracted if its enemy pieces)
2. Pawn advancement: a score that becomes higher the more pawns a player has advanced up the board
3. Checkmate: a hugely high score that represents a checkmate and is designed to make all other scoring redundant

Flowchart:



Simulate Move

This function will make a passed move on a passed board and return the new board.

It takes 2 parameters:

- move = the move to be carried out (array)
- board = the board the move should be carried out on (2d array)

Pseudocode:

```
new_board ← copy.deepcopy(board)
IF move[2] ≠ None THEN
    new_board[move[3]][move[2]] ← new_board[move[1]][move[0]]
    new_board[move[1]][move[0]] ← None

new_board[move[3]][move[2]].change_location(calculate_coordinates(x=move[2],
y=move[3]))

    IF (move[3] = 0 OR move[3] = 7) AND new_board[move[3]][move[2]].get_type()
= 'P' THEN
        new_board[move[3]][move[2]].promote('Q')
    ENDIF
ELSE
    IF move[0] = 0 THEN
        new_board[move[1]][3] ← new_board[move[1]][move[0]]
        new_board[move[1]][3].change_location('d' + abs(move[1]-8))
        new_board[move[1]][move[0]] ← None
```

```
        new_board[move[1]][2] ← new_board[move[1]][4]
        new_board[move[1]][2].change_location('c' + abs(move[1]-8))
        new_board[move[1]][4] ← None
    ELSE
        new_board[move[1]][5] ← new_board[move[1]][move[0]]
        new_board[move[1]][5].change_location('f' + abs(move[1]-8))
        new_board[move[1]][move[0]] ← None
        new_board[move[1]][6] ← new_board[move[1]][4]
        new_board[move[1]][6].change_location('g' + abs(move[1]-8))
        new_board[move[1]][4] ← None
    ENDIF
ENDIF
RETURN new_board
```

Random Move

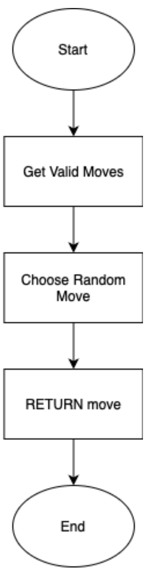
The random move works by calculating all possible moves (only for the computers next turn) and choosing one at random to play.

This makes up the first level of competency the AI has and will be used for the Easy difficulty level.

The function will return the move in the form:
[startx,starty,endx,endy]

The function should only take 2 parameters:
1. The board to use (2d array)
2. The board object (object)

Flowchart:



Basic Move

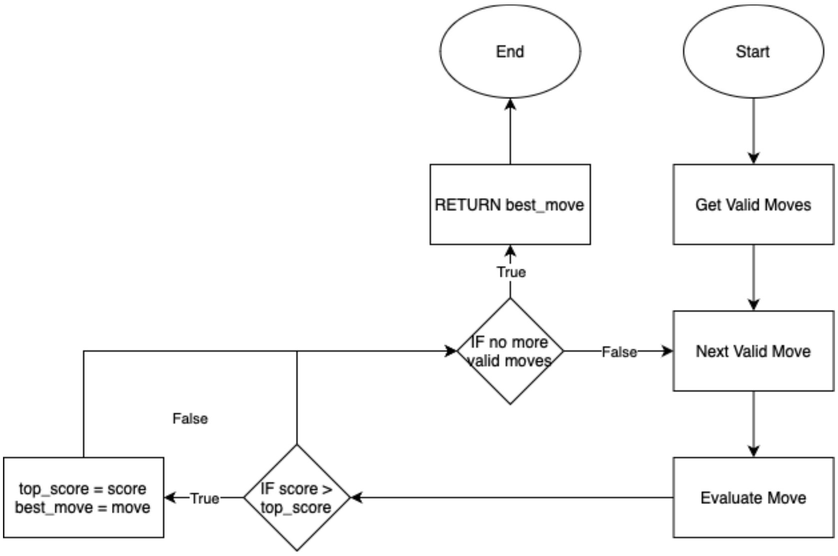
The basic move works by calculating all possible moves (only for the computers next turn) and then choosing the one that has the best direct outcome (using the board evaluation function).

This makes up the second level of competency of the AI and will be the Med difficulty level.

The function will return the move in the form:
[startx,starty,endx,endy]

The function should only take 2 parameters:
1. The board to use (2d array)
2. The board object (object)

Flowchart:



Advanced Move

The advanced move works by managing the minmax; it calls the minmax for the first ply with the specified total depth of the search and -/+ infinity for the alpha and beta values respectively.

This makes up the third and final level of competency of the AI and will be the Hard difficulty level.

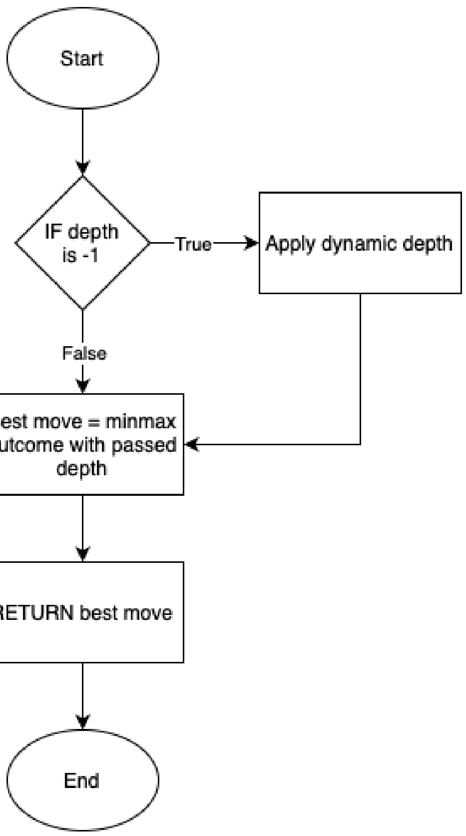
The function will return the move in the form:
[startx,starty,endx,endy]

The function should only take 2 parameters:

1. The board to use (2d array)
2. The board object (board object)

The function also has the option for a simple dynamic depth. This is when the depth is set to a higher value if there are less initial valid moves. This will activate when the specified total depth is '-1'.

Flowchart:



Minmax

The minmax function is a vital part to the computers AI and is used for the advanced move function.

It is used to calculate the best move by looking at possible board outcomes from the next few possible moves and choosing the best outcome for the computer (given the opposition will play the best moves possible).

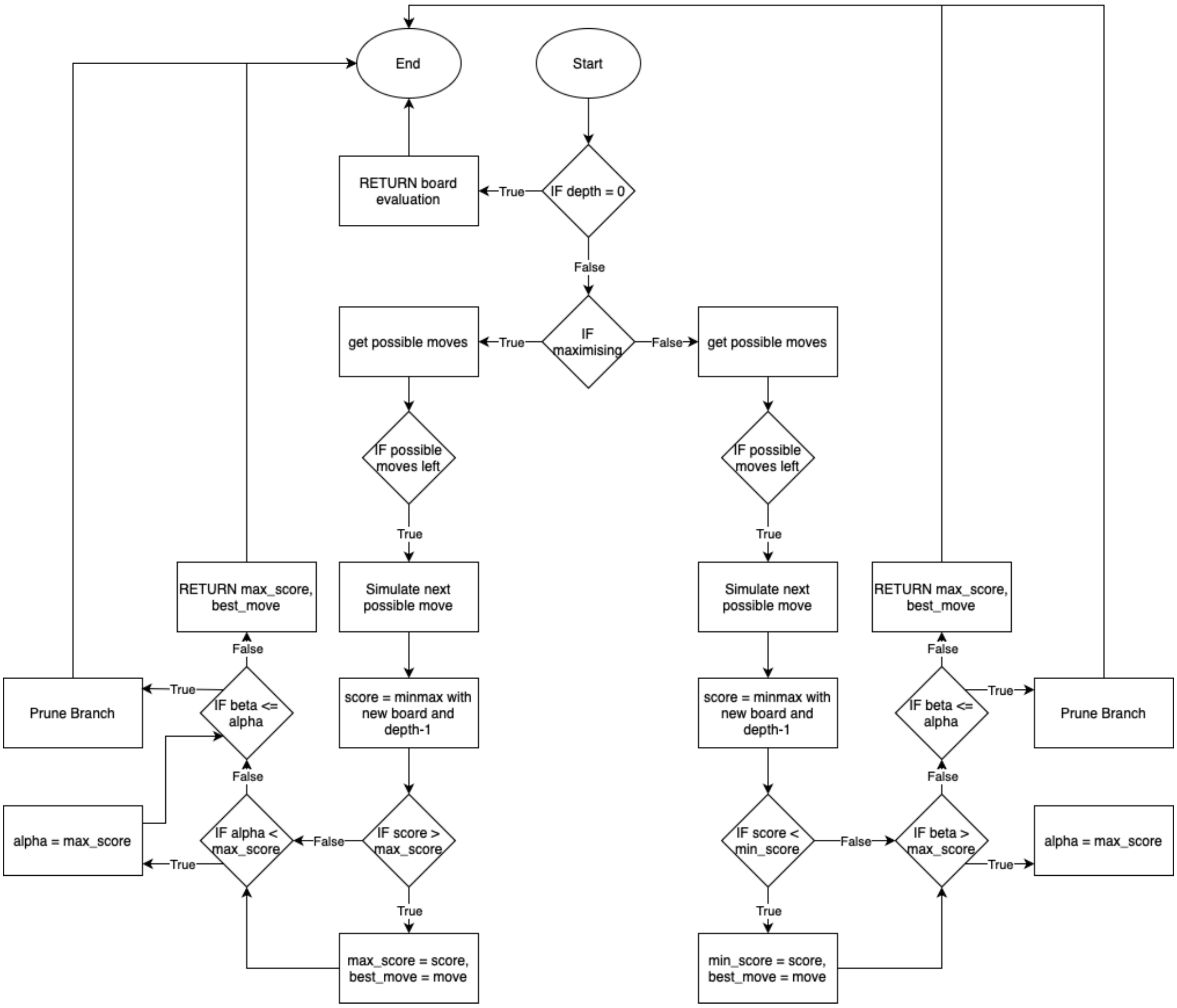
The minmax function is a recursive algorithm that calls itself every time it needs to predict the next best move. The possible moves are calculated for a specified number of future move (the depth). The minmax should take a much longer amount of time for higher depths as the number of boards evaluated increases exponentially with the depth.

The minmax will also make use of alpha beta pruning which is used to prune branches of the minmax that are already guaranteed to yield the worst results. The alpha and beta values keep track of the worst and best moves for the player so far which are used to identify and prune irrelevant branches before they are fully checked. This majorly decreases the minmax search time.

To support the alpha beta pruning, the minmax makes use of the move ordering from the get valid moves function. This orders the moves based on their immediate outcome which is a very rough prediction of how the whole branch will end out. This means more promising moves are checked by the minmax first, so the alpha beta pruning is likely to quickly prune later branches.
The minmax function will take 6 parameters:

1. board_to_use = the board the minmax (2d array)
2. board = the board object (object)
3. depth = the depth of the minmax search (how many moves it looks into the future) (integer)
4. alpha = the value of alpha (integer)
5. beta = the value of beta (integer)
6. max_or_min = weather the minmax should be maximising or minimising the score (this will alternate) (boolean)

Flowchart:



End
Start
RETURN board evaluation
True
IF depth = 0
False
get possible moves
True
IF maximising
False
get possible moves
IF possible moves left
True
Simulate next possible move
score = minmax with new board and depth-1
IF score > max_score
False
True
max_score = score, best_move = move
IF alpha < max_score
True
alpha = max_score
False
IF beta <= alpha
True
Prune Branch
False
RETURN max_score, best_move
IF possible moves left
True
Simulate next possible move
score = minmax with new board and depth-1
IF score < min_score
False
True
min_score = score, best_move = move
IF beta > max_score
True
alpha = max_score
False
IF beta <= alpha
True
Prune Branch
False
RETURN max_score, best_move

Testing

IO Testing

UI Button Clicks

Load from Previous Game (1.1)

VIDEO: https://www.youtube.com/watch?v=KmJbw75-lh0
TIMESTAMP: 00:00

This series of tests is to make sure the button click functionality and implementation is working for the load previous game prompt.

The button press should dictate weather the value of a variable should be changed to True of False. This variable dictates weather the previous game should be loaded.

'Yes' Button

Num.	**Description**	**Test**	**Expected Outcome**	**Outcome + timestamp**
1.1.1	Check that clicking the 'Yes' button directly works (typical)	Click the 'Yes' button in the middle	The value of the variable that indicates if the previous game should be loaded should be True	00:21 Same as expected outcome
1.1.2	Check that clicking just inside the upper boundary of the 'Yes' button works (extreme)	Click the 'Yes' button just inside the upper boundary	The value of the variable that indicates if the previous game should be loaded should be True	00:46 Same as expected outcome
1.1.3	Check that clicking just inside the lower boundary of the 'Yes' button works (extreme)	Click the 'Yes' button just inside the lower boundary	The value of the variable that indicates if the previous game should be loaded should be True	00:59 Same as expected outcome
1.1.4	Check that clicking just inside the left boundary of the 'Yes' button works (extreme)	Click the 'Yes' button just inside the left boundary	The value of the variable that indicates if the previous game should be loaded should be True	01:07 Same as expected outcome

1.1.5	Check that clicking just inside the right boundary of the 'Yes' button works (extreme)	Click the 'Yes' button just inside the right boundary	The value of the variable that indicates if the previous game should be loaded should be True	01:18 Same as expected outcome
1.1.6	Check that clicking just outside the upper boundary of the 'Yes' button works (erroneous)	Click the 'Yes' button just outside the upper boundary	Nothing should happen and the window should not close. The variable that indicates the previous game should be loaded should be False	01:26 Same as expected outcome
1.1.7	Check that clicking just outside the lower boundary of the 'Yes' button works (erroneous)	Click the 'Yes' button just outside the lower boundary	Nothing should happen and the window should not close. The variable that indicates the previous game should be loaded should be False	01:44 Same as expected outcome
1.1.8	Check that clicking just outside the left boundary of the 'Yes' button works (erroneous)	Click the 'Yes' button just outside the left boundary	Nothing should happen and the window should not close. The variable that indicates the previous game should be loaded should be False	01:51 Same as expected outcome
1.1.9	Check that clicking just outside the right boundary of the 'Yes' button works (erroneous)	Click the 'Yes' button just outside the right boundary	Nothing should happen and the window should not close. The variable that indicates the previous game should be loaded should be False	01:58 Same as expected outcome

‘No’ Button

The code for the ‘No’ button click detection is the same as the code for the ‘Yes’ button and therefore the TEX (typical erroneous extreme) tests for the ‘Yes’ button can represent the TEX tests for the ‘No’ button.

Num.	Description	Test	Expected Outcome	Outcome + timestamp
1.1.10	Check that clicking the ‘No’ button works	Click the ‘No’ button	The value of the variable that indicates if the previous game should be loaded should be False. The window should close	02:04 Same as expected outcome

Choose Difficulty Level (1.2)

VIDEO: https://www.youtube.com/watch?v=KmJbw75-lh0
TIMESTAMP: 02:32

This series of tests is to make sure the difficulty selection button click functionality and implementation is working.

The button press should alter the value of a difficulty variable. This should be ‘0’ for ‘Easy’, ‘1’ for ‘Med’ and ‘2’ for ‘Hard’.

‘Easy’ Button

The code for the ‘Easy’ button click detection is the same as the code for the ‘Yes’ button (1.1.1 - 1.1.9) and therefore the TEX (typical erroneous extreme) tests for the ‘Yes’ button can represent the TEX tests for the ‘Easy’ button.

Num.	Description	Test	Expected Outcome	Outcome + timestamp
1.2.1	Check that clicking the ‘Easy’ button works	Click the ‘Easy’ button	The value of the difficulty variable should be 0. The window should close	02:57 Same as expected outcome

‘Med’ Button

The code for the ‘Med’ button click detection is the same as the code for the ‘Yes’ button (1.1.1 - 1.1.9) and therefore the TEX (typical erroneous extreme) tests for the ‘Yes’ button can represent the TEX tests for the ‘Med’ button.

Num.	Description	Test	Expected Outcome	Outcome + Timestamp
1.2.2	Check that clicking the ‘Med’ button works	Click the ‘Med’ button	The value of the difficulty variable should be 1. The window should close	03:33 Same as expected outcome

‘Hard’ Button

The code for the ‘Hard’ button click detection is the same as the code for the ‘Yes’ button (1.1.1 - 1.1.9) and therefore the TEX (typical erroneous extreme) tests for the ‘Yes’ button can represent the TEX tests for the ‘Hard’ button.

Num.	Description	Test	Expected Outcome	Outcome + Timestamp
1.2.2	Check that clicking the ‘Hard’ button directly works	Click the ‘Hard’ button in the middle	The value of the difficulty variable should be 2. The window should close	03:46 Same as expected outcome

Promote Piece (1.3)

VIDEO: https://www.youtube.com/watch?v=KmJbw75-lh0
TIMESTAMP: 04:01

This series of tests is to make sure the promotion selection button click functionality and implementation is working.

The button press should affect what piece a pawn is promoted to.

This test will be of a very similar nature to the previous tests however the outcome should be based on whether the pawn promotes and if it’s to the correct piece.

This test will only test for the functionality of the result of clicking the button and will not test the actual clicking as the code that handles the clicking is the same as the code tested in the previous tests.

Num.	Description	Test	Expected Outcome	Outcome + Timestamp
1.3.1	Check the promote to knight button is working	Move a pawn to the furthest rank and click the knight button	The menu should close and the pawn should promote into a knight	04:25 Same as expected outcome
1.3.2	Check the promote to queen button is working	Move a pawn to the furthest rank and click the queen button	The menu should close and the pawn should promote into a queen	04:42 Same as expected outcome

Select Board Square (1.4)

VIDEO: https://www.youtube.com/watch?v=KmJbw75-lh0
TIMESTAMP: 04:54

This is a series of tests that will make sure the program is handling the pressing of board squares by the user.

Each button press could affect what pieces are being selected and where pieces are being moved to.

This test will also be of a very similar nature to the previous tests however the outcome should be based on board square highlights and pieces moving.

This test will only test for the functionality of the result of clicking the board and will not test the actual clicking as the code that handles the clicking is the same as the code tested in the previous tests.

Num.	Description	Test	Expected Outcome	Outcome + Timestamp
1.4.1	Test the correct board square is selected when clicked (with a piece on)	Select any piece belonging to you	The square the piece is on should be highlighted green	04:57 Same as expected outcome
1.4.2	Test the correct board square is selected when clicked (with no piece)	Select a piece followed by a square it can legally move to	The piece should move to the clicked square and the square should be highlighted red	05:31 Same as expected outcome

Board Save File (2.1)

VIDEO: https://www.youtube.com/watch?v=algWmoaRpqk
TIMESTAMP: 00:00

This series of tests is to make sure the code reads and implements the data from the board save file correctly. The program should handle erroneous data by ending the program after displaying an error message.

Num.	Description	Test Data	Expected Outcome	Outcome + Timestamp
2.1.1	Test the file input with normal data (including boundary data for coordinates on all sides) - the starting board with 64 squares	W [R a8 B] [N b8 B] [B c8 B] [Q d8 B] [K e8 B] [B f8 B] [N g8 B] [R h8 B] [P a7 B] [P b7 B] [P c7 B] [P d7 B] [P e7 B] [P f7 B] [P g7 B] [P h7 B] [None] [P a2 W] [P b2 W] [P c2 W] [P d2 W] [P e2 W] [P f2 W] [P g2 W] [P h2 W] [R a1 W] [N b1 W] [B c1 W] [Q d1 W] [K e1 W] [B f1 W] [N g1 W] [R h1 W]	The board should load into a state that resembles how a chess game starts according to chess rules. The white player should have the next move	00:10 Same as expected outcome
2.1.2	Test the file input with reduced data - the starting board with only pieces (no empty squares)	B [R a8 B] [N b8 B] [B c8 B] [Q d8 B] [K e8 B] [B f8 B] [N g8 B] [R h8 B] [P a7 B] [P b7 B] [P c7 B] [P d7 B] [P e7 B] [P f7 B] [P g7 B] [P h7 B] [P a2 W] [P b2 W] [P c2 W] [P d2 W] [P e2 W] [P f2 W] [P g2 W] [P h2 W] [R a1 W] [N b1 W] [B c1 W] [Q d1 W] [K e1 W] [B f1 W] [N g1 W] [R h1 W]	The board should load into a state that resembles how a chess game starts according to chess rules. The black player should have the next move	00:25 Same as expected outcome

2.1.3	Test the file input with erroneous location data - the starting board with a piece that has a location out of range (first rook listed)	W [R i9 B] [N b8 B] [B c8 B] [Q d8 B] [K e8 B] [B f8 B] [N g8 B] [R h8 B] [P a7 B] [P b7 B] [P c7 B] [P d7 B] [P e7 B] [P f7 B] [P g7 B] [P h7 B] [None] [P a2 W] [P b2 W] [P c2 W] [P d2 W] [P e2 W] [P f2 W] [P g2 W] [P h2 W] [R a1 W] [N b1 W] [B c1 W] [Q d1 W] [K e1 W] [B f1 W] [N g1 W] [R h1 W]	The program should exit with an error message stating there was an error creating the piece / coordinates are invalid	00:52 Same as expected outcome
2.1.4	Test the file input with erroneous location data - the starting board with a piece that has a location with a string in place of the number (first rook listed)	W [R ab B] [N b8 B] [B c8 B] [Q d8 B] [K e8 B] [B f8 B] [N g8 B] [R h8 B] [P a7 B] [P b7 B] [P c7 B] [P d7 B] [P e7 B] [P f7 B] [P g7 B] [P h7 B] [None] [P a2 W] [P b2 W] [P c2 W] [P d2 W] [P e2 W] [P f2 W] [P g2 W] [P h2 W] [R a1 W] [N b1 W] [B c1 W] [Q d1 W] [K e1 W] [B f1 W] [N g1 W] [R h1 W]	The program should exit with an error message stating there was an error creating the piece / coordinates are invalid	01:16 Same as expected outcome
2.1.5	Test the file input with erroneous colour data - the starting board with a piece that has an incorrect colour identifier (first rook listed)	W [R a1 P] [N b8 B] [B c8 B] [Q d8 B] [K e8 B] [B f8 B] [N g8 B] [R h8 B] [P a7 B] [P b7 B] [P c7 B] [P d7 B] [P e7 B] [P f7 B] [P g7 B] [P h7 B] [None] [P a2 W] [P b2 W] [P c2 W] [P d2 W] [P e2 W] [P f2 W] [P g2 W] [P h2 W] [R a1 W] [N b1 W] [B c1 W] [Q d1 W] [K e1 W] [B f1 W] [N g1 W] [R h1 W]	The program should exit with an error message stating there was an error creating the piece / colour is invalid	01:37 Same as expected outcome

2.1.6	Test the file input with erroneous piece type data - the starting board with a piece that has an incorrect piece identifier (first rook listed)	W [E a1 B] [N b8 B] [B c8 B] [Q d8 B] [K e8 B] [B f8 B] [N g8 B] [R h8 B] [P a7 B] [P b7 B] [P c7 B] [P d7 B] [P e7 B] [P f7 B] [P g7 B] [P h7 B] [None] [P a2 W] [P b2 W] [P c2 W] [P d2 W] [P e2 W] [P f2 W] [P g2 W] [P h2 W] [R a1 W] [N b1 W] [B c1 W] [Q d1 W] [K e1 W] [B f1 W] [N g1 W] [R h1 W]	The program should exit with an error message stating there was an error creating the piece / piece type is invalid	01:56 Same as expected outcome
2.1.7	Test the file input with only the letter indicating who has the next move.	W	The program should exit with an error message stating the file was empty	02:16 Same as expected outcome
2.1.8	Test the file input with nothing in it		The program should exit with an error message stating the file was empty	02:34 Same as expected outcome

White-Box Testing

Minmax (3.1)

This will step through the minmax function to check for multiple key parts of its functionality.

The depth of the minmax will always be 2 unless specified otherwise.

VIDEO: https://www.youtube.com/watch?v=C4zf_g88B7I

3.1.1 The minmax is choosing the correct best / worst move
TIMESTAMP: 00:14

Test	Description	Expected Outcome	Outcome (+timestamp)
3.1.1.1	Use a print statement to display the current move score, the highest move score, and the best move for the first depth in a minmax search.	When the current move score is greater than the best move score, the best move score should be updated to the current move score value and the best move should be updated to the current move. If the current move score is not greater than the best move score, nothing should change in that pass.	00:14 Same as expected outcome
3.1.1.2	Use a print statement to display the current move score, the worst move score, and the worst move for the second depth in a minmax search (one set of valid moves only)	When the current move score is lower than the worst move score, the worst move score should be updated to the current move score value and the worst move should be updated to the current move. If the current move score is not lower than the worst move score, nothing should change in that pass.	05:20 Same as expected outcome

3.1.1 Gives evidence for the minmax choosing the best/worst move depending on whether its maximising or minimising.

3.1.2 The alpha beta pruning is pruning a branch when it is deemed redundant TIMESTAMP: 09:27

Test	Description	Expected Outcome	Outcome (+timestamp)
3.1.2.1	Use a print statement to display the current beta value, the current alpha value and when a branch is pruned	Then the beta value is <= the alpha value, the branch should be pruned since it is guaranteed to yield the worst results	09:27 Same as expected outcome

3.1.2 Gives evidence that the alpha beta pruning is working by showing examples of when branches are pruned and checking that they should be pruned at those times.

3.1.3 All the possible valid moves are being looped through (until there is reason to terminate)
TIMESTAMP: 12:38

Test	Description	Expected Outcome	Outcome (+timestamp)
3.1.3.1	Use print statements to show a list of the valid moves, and the move the loop is currently on for the first depth in a minmax search	Each move in the valid moves list should be run through on the loop. This means every move is considered.	12:38 Same as expected outcome

3.1.3 Proves that the minmax function is checking all valid moves on the board (the valid moves function it uses is working).

3.1.4 The minmax is going to the correct depth
TIMESTAMP: 14:39

Test	Description	Expected Outcome	Outcome (+timestamp)
3.1.4.1	Use a print statement to display the current depth the function is on when running the minmax at a depth of 2	The highest number printed should be 2, and the lowest number should be 0	14:39 Same as expected outcome
3.1.4.2	Use a print statement to display the current depth the function is on when running the minmax at a depth of 3	The highest number printed should be 3, and the lowest number should be 0	15:58 Same as expected outcome

3.1.4 Proves the minmax is going to the correct specified depth

Board Evaluation (3.2)

This will check that the outcome of the board evaluation is correct for the board its evaluating.
TIMESTAMP: 16:36

Test	Description	Expected Outcome	Outcome (+timestamp)
3.2.1	Get a board state with no checks or checkmates and print out its evaluation score using the function	The evaluation output should match what the evaluation score is when manually calculated	16:36 Same as expected outcome
3.2.2	Get a board state with a checkmate and print out its evaluation score using the function	The equation output should be in the thousands (positive or negative)	20:31 Same as expected outcome

3.2 Proves the board evaluation function is evaluating the board correctly.

Requirement Testing (4)

LINK: https://www.youtube.com/watch?v=3HIwX6EZgdg
TIMESTAMP: 00:00

This series of tests will run through the requirements and make sure each one is met. The requirements not tested here are more subjective and will be tested in the beta testing (5.1) or they have already been tested.

Num.	Requirement	Test	Expected outcome	Outcome + Timestamp
4.1.1	1.1.2	White should make a move and wait for the computer to make a move	The board should be automatically edited according to the move the computer calculates	00:17 Same as expected outcome
4.1.2	1.1.1	White should make a move and the computers next move should be observed	The move the board is edited to show (that the computer calculated) should follow all the rules of chess	01:02 Same as expected outcome
4.1.3	1.2.1	The computer should be set to its first level of competency and observed handling the same scenario 3 times	The move the computer makes should not be consistent through the 3 times (it should be random)	01:21 Same as expected outcome
4.1.4	1.2.2	The computer should be set to the second level of competency and observed handling the same scenario 3 times	The move the computer plays should be consistent through the 3 moves by putting itself into the highest scoring position immediately after the move is played	02:13 Same as expected outcome

4.1.5	1.2.3	The computer should be set to the third level of competency and observed handling the same move 3 times	The move the computer plays should be consistent through the 3 moves and have an output determined by the minmax function (tested in the white box testing)	03:11 Same as expected outcome
4.1.6	1.3	Time how long the computer takes to make its move for 5 different moves on each competency level	The time should be less than 1 min for all moves	04:02 Same as expected outcome
4.2.1	2.1.1 & 2.2	The easy difficulty level should be selected at the start	The computer should make moves with its lowest level of competency (random moves) and the stored difficulty should be '0'	05:19 Same as expected outcome
4.2.2	2.1.2 & 2.2	The med difficulty level should be selected at the start	The computer should make moves with its medium level of competency (looking at the scores for its move) and the stored difficulty should be '1'	06:14 Same as expected outcome
4.2.3	2.1.3 & 2.2	The hard difficulty level should be selected at the start	The computer should make moves with its highest level of competency (looking multiple moves into the future) and the stored difficulty should be '2'	06:28 Same as expected outcome
4.3.1	3.1	Observe the board layout	The dimensions of the board should be 8 x 8 squares	06:41 Same as expected outcome

4.3.2	3.2	Observe the board side labelling	The x-axis should be labelled in letters in the order of the alphabet from left to right. The y-axis should be labelled in numbers going from bottom to top starting with 1	06:55 Same as expected outcome
4.3.3	3.5	Finish the user input menus at the start of the game with the board not loaded and any difficult and observe the piece positioning	The white pieces should be ordered with all pawns on the second rank up, and the following order from left to right on the bottom rank: rook, knight, bishop, queen, king, bishop, knight, rook. This should be mirrored for the black pieces at the other end.	07:12 Same as expected outcome
4.4.1	4.1.1	Move a pawn in a legal way. (one forwards when there is nothing in front)	The pawn should be moved and the piece selections should be cleared	07:22 Same as expected outcome
4.4.2	4.1.1	Move a pawn two spaces forward when it's on its starting rank and it's not moving through a piece or onto a piece	The pawn should be moved and the piece selections should be cleared	07:39 Same as expected outcome
4.4.3	4.1.1	Move a pawn diagonally onto a square with an enemy piece on	The pawn should be moved, the piece selections should be cleared and the piece at the end location should be removed	07:52 Same as expected outcome
4.4.4	4.1.1	Move a pawn in an illegal way. (three forwards when there is nothing in front)	The pawn should not move and the piece selections should be cleared	08:16 Same as expected outcome

4.4.5	4.1.1	Try to move diagonally with a pawn when there is no piece to take (and no en passant)	The pawn should not move and the piece selections should be cleared	08:36 Same as expected outcome
4.4.6	4.1.1	Try to move a pawn forward two when it's not on its starting rank	The pawn should not move and the piece selections should be cleared	08:49 Same as expected outcome
4.4.7	4.1.1	Move a rook horizontally any amount but not through a piece and not taking a piece.	The rook should be moved and the piece selections should be cleared	09:00 Same as expected outcome
4.4.8	4.1.1	Move a rook vertically any amount but not through a piece and not taking a piece	The rook should be moved and the piece selections should be cleared	09:20 Same as expected outcome
4.4.9	4.1.1	Move a rook to any horizontal location from which the path is blocked by another piece or occupied	The rook should not be moved and the piece selections should be cleared	09:37 Same as expected outcome
4.4.10	4.1.1	Attempt to move a rook diagonally	The rook should not be moved and the piece selections should be cleared	09:29 Same as expected outcome
4.4.11	4.1.1	Move a knight in all 8 legal ways where the final space isn't occupied	The knight should be moved and the piece selections should cleared all 8 times	09:54 Same as expected outcome
4.4.12	4.1.1	Move a knight in an illegal way	The knight should not be moved and the piece selections should be cleared	10:58 Same as expected outcome

4.4.13	4.1.1	Move a bishop diagonally any amount to any position that isn't blocked by any piece or occupied	The bishop should be moved and the piece selections should be cleared	11:13 Same as expected outcome
4.4.14	4.1.1	Move a bishop to any diagonal location from which the path is blocked by another piece	The bishop should not be moved and the piece selections should be cleared	11:37 Same as expected outcome
4.4.15	4.1.1	Move a queen horizontally and vertically any amount to a position that isn't blocked or occupied	The queen should be moved and the piece selections should be cleared	12:03 Same as expected outcome
4.4.16	4.1.1	Move a queen diagonally any amount to a position that isn't blocked or occupied	The queen should be moved and the piece selections should be cleared	12:38 Same as expected outcome
4.4.17	4.1.1	Move a queen non diagonally horizontally or vertically to a position that isn't blocked or occupied	The queen should not be moved and the piece selections should be cleared	12:51 Same as expected outcome
4.4.18	4.1.1	Move a king in all possible directions (legally)	The king should be moved every time and the piece selections cleared	13:13 Same as expected outcome
4.4.19	4.1.1	Move a king two spaces away from where it is (which is an illegal move)	The king should not be moved and the piece selections should be cleared	13:38 Same as expected outcome
4.4.20	4.1.2	Attempt to move a piece in a way that puts your own king in check	The piece should not move, and the piece selections should clear	13:53 Same as expected outcome

4.4.21	4.1.3	Move any piece onto a square with an enemy piece	The piece should move and the enemy piece should be replaced by the moved piece	14:38 Same as expected outcome
4.4.22	4.2.1	Use en passant with a pawn when it is legal	The pawn should move to the selected square and the pawn that the en passant is being carried out against should be removed	15:20 Same as expected outcome
4.4.23	4.2.2	Attempt to castle with blocking pieces in-between the chosen rook and the king	The pieces should not move, and the piece selections should clear	16:11 Same as expected outcome
4.4.24	4.2.2	Attempt to castle while the king is in check	The pieces should not move, and the piece selections should clear	16:46 Same as expected outcome
4.4.25	4.2.2	Attempt to castle after the king and rook have been moved	The pieces should not move, and the piece selections should clear	17:11 Same as expected outcome
4.4.26	4.2.2	Attempt to castle when it is legal to do so	The pieces should be moved in the castling nature	17:59 Same as expected outcome
4.4.27	4.2.3	Move a pawn to the furthest rank and attempt to promote to a queen	The promotion choice screen should appear, and the pawn should turn into a queen	18:36 Same as expected outcome
4.4.28	4.2.3	Move a pawn to the furthest rank and attempt to promote to a knight	The promotion choice screen should appear, and the pawn should turn into a knight	19:02 Same as expected outcome
4.4.29	4.3	Make any legal move	The board UI should change to show the move that has just been made	19:16 Same as expected outcome

4.5.1	5.1	Make a print statement to output if the king is in check. Make sure the king is not in check.	There should be no output indicating the king is not in check	19:46 Same as expected outcome
4.5.2	5.1	Make a print statement to output if the king is in check. Make sure the king is in check	There should be some output indicating the king is in check	20:39 Same as expected outcome
4.5.3	5.2	Create a checkmate scenario where a king is in checkmate	The game should end and there should be a notification stating the king was in checkmate	21:36 Same as expected outcome
4.5.4	5.3	Create a stalemate scenario	The game should end in a draw and the user should be notified	22:16 Same as expected outcome
4.6.1	6.1.1	Save a board state, view the save file and manually it back to a visual representation of the board	The board saved and the manual visual representation should match	22:44 Same as expected outcome
4.6.2	6.1.2	Save a board state and view the save file	The save file should clearly state who's turn it is next at the start	24:15 Same as expected outcome
4.6.3	6.2	Start a game and wait for the prompt asking if you want to load the previous game	The prompt should show before anything else	24:40 Same as expected outcome
4.6.4	6.1	Select load game when asked if you want to load the previous game	The exact previous game state should be loaded and the player who's turn it was next should have the next turn	24:56 Same as expected outcome

Beta Testing (5.1)

This testing stage will involve having a player play the game and give feedback on it.

The player I will be using is Will Ward who is the chess player I originally interviewed in my research.

Will is going to play a game of chess against each of the difficulty levels and give feedback using a series of prompts:

- The speed of the AI calculation (requirement 1.3)
- The user friendliness + graphical user interface of the game (requirements 3.3, 3.4)
- The ranking of the difficulty levels (requirement 2.1)
- The ability of the AI (requirement 1.2)

Will is also going to explore the load previous game feature and provide feedback using another series of prompts:

- The accuracy of the save (requirement 6.1)
- The ease of loading the save (requirement 6.2)

Answers

Test	Prompt	Response	Evaluation
5.1.1	Speed of calculation	"Both the easy and medium difficulty levels were pretty much instant which is amazing. The hard difficulty level took a few seconds and varied from move to move however it was generally around the same time a human would take which is acceptable and even seems more realistic"	Will found the speed of the calculation to be acceptable and even thought it made it more realistic.
5.1.2	User friendliness	"The game is very intuitive and easy to use. The prompts with buttons are very easy to understand and respond to and the interface design isn't too crowded. It's very easy to see and understand the current board state with the location and types of the pieces and making moves with the pieces is simple and easy."	Will found the UI very easy and intuitive to use. He said the location and type of pieces was very easy to understand and the movement of pieces was simple.

5.1.3	Ranking of difficulty levels	“The difficulty levels are ranked well and easily distinguishable as different levels of competency with easy being simple to beat medium being slightly more challenging and hard being much trickier.”	Will found the difficulty levels to be ranked well and could easily tell the difference between them.
5.1.4	Ability of the AI	“I find the easy and medium levels simple to beat however I think they would be useful for people with less chess skill than me. When playing the hard difficulty I experience much more of a challenge.”	Wil found the AI hard difficulty gives him a bit of a challenge, and thought easy and med levels were simple to beat and would suit someone of lower skill.
5.1.5	Accuracy of the board save	“When I load the board is perfectly matches what I left it as. The game will even know who’s turn it should be next”	Will concluded that the board save was saving accurately with both the board state and the next player.
5.1.6	Ease of loading the save	“There is a really simple prompt with a simple button to press to load the game.”	Will thought the board load feature was simple to use.

Evaluation

Requirements

The testing is organised by requirement and evidence for the requirements being met can be seen there.

Requirement	**Description**	**Met? (+where)**	**Evaluation**
1.1.1	The CBP must make its moves as per the rules of chess	Yes Test 4.1.2	The test shows how the computers move follows the correct rules of chess (by showing an example). This requirement, therefore, is fully met
1.1.2	The CBP must make its move automatically when its on its turn	Yes Test 4.1.1	The test shows how the CBP starts calculating and makes its move automatically after the user (white) has made theirs. This requirement, therefore, was fully met
1.2.1	First level of competency: make random moves	Yes Tests 4.1.3, 5.1.4	4.1.3 checks that the moves the computer makes at this level of competency are inconsistent. This shows that they are random therefore proving this requirement is fully met. 5.1.4 shows how the levels of competency differ from each other in the intended way in a beta testing scenario

1.2.2	Second level of competency: make the best move for that go (without looking ahead)	Yes Tests 4.1.4, 5.1.4	4.1.4 shows how the outcome score of each move the computer makes is consistent. This proves how the computer is evaluating the end positions of its valid moves and choosing the highest scoring one consistently. This, therefore, proves the requirement is fully met. 5.1.4 shows how the levels of competency differ from each other in the intended way in a beta testing scenario
1.2.3	Third level of competency: make the best move for that go (considering moves ahead)	Yes Tests 4.1.5, 3.1, 5.1.4	4.1.5 Shows how the outcome the computer comes up with is consistent which means it is always using the same minmax. The 3.1 white box tests prove individually for each element of the minmax that the minmax is functioning correctly. The white box tests are evaluated individually in the testing section. 5.1.4 shows how the levels of competency differ from each other in the intended way in a beta testing scenario
1.3	The computer must make its move in reasonable time (1min)	Yes Tests: 4.1.6, 5.1.1	4.1.6 proves the highest processing time is still under 1 min consistently. 5.1.1 proves the speed of the AI calculation is acceptable in a beta testing scenario.

2.1.1	0 difficulty = easiest to beat (easy difficulty level)	Yes Tests: 4.2.1, 5.1.3	4.2.1 proves that the correct level of competency is selected for the easy difficulty level with is the random moves which is the easiest to beat. 5.1.3 Proves the difficulty levels are ranked correctly in a beta testing scenario.
2.1.2	1 difficulty = med to beat (med difficulty level)	Yes Tests: 4.2.2, 5.1.3	4.2.1 proves that the correct level of competency is selected for the med difficulty level with is the best move for its direct outcome which is the middle difficulty to beat. 5.1.3 Proves the difficulty levels are ranked correctly in a beta testing scenario.
2.1.3	2 difficulty = hardest to beat (hard difficulty level)	Yes Tests: 4.2.3, 5.1.3	4.2.1 proves that the correct level of competency is selected for the hard difficulty level with is the random moves which is the hardest to beat. 5.1.3 Proves the difficulty levels are ranked correctly in a beta testing scenario.
2.2	Difficulty setting should be retrieved from the user at the start using GUI window	Yes Tests: 4.2.1-3	These tests prove that the UI window displayed at the start of the game is functioning correctly and can be properly used to retrieve the difficulty level.
3.1	Should be correct 8x8 chess board structure with 64 squares	Yes Test: 4.3.1	This test proves the board layout is correct according to the design and requirement
3.2	The board should be labelled with letters on the x-axis and numbers on the y-axis according to the general chess board coordinate layout	Yes Test: 4.3.2	This test proves the board labelling is correct according to the design and requirement.

3.3	Should be obvious when a piece is in a square (and what square the piece is in)	Yes Test 5.1.2	This test proves it's obvious what square pieces are in as the user in the beta test describes it as 'very easy' to understand the locations of the pieces were
3.4	Each piece should be easy to identify according to the accepted general appearance of each chess piece	Yes Test 5.1.2	This test proves it's obvious what each piece type is as the user in the beta test describes it as 'very easy' to understand the types of pieces
3.5	The piece layout at the start of the game should be correct according to the rules of chess	Yes Test: 4.3.3	This test proves the board starts with the correct board layout as shown in the test video.
4.1.1	Each piece should only be moved based on what the rules of chess dictate it can do	Yes Tests: 4.4.1-19	These tests individually test the movement of each piece using valid and erroneous data. This proves that each piece is correctly movable according to the rules of chess as researched. The tests match the requirement.
4.1.2	A player cannot move itself into check	Yes Test: 4.4.20	This test proves how a piece can't be moved if it involves putting/keeping your king in check which matches the requirement.
4.1.3	If a move ends on an enemy piece, the enemy piece should be deleted (taken)	Yes Test:4.4.21	This test proves enemy piece taking is working as expected which matches the requirement.
4.2.1	En passant can only occur (and should be possible) when the specific chess rules for en passant are met	Yes Test: 4.4.22	This test proves that en passant is functioning correctly in the game which matches the requirement.
4.2.2	Castling can only occur (and should be possible) when the specific chess rules for Castling are met	Yes Tests: 4.4.23-26	These tests prove the castling function is correctly working by testing the responses to erroneous and correct data. The tests match the requirement.

4.2.3	Promotion can only occur (and should be possible) when the specific chess rules for promotion are met	Yes Tests: 4.4.27-28	These tests individually prove promoting to a queen and to a knight both work which matches the requirement.
4.3	When a piece is moved, the new board state should be shown on the displayed board	Yes Tests: 4.4.29	This tests proves the move making system is working properly and moves are displayed to the user. This matches the requirement.
5.1.1	If a king is threatened by an opponent's piece, it in in check	Yes Tests: 4.5.1-2	These tests validate that the game correctly identifies when a king is and isn't in check. This fulfils the requirement.
5.2.1	If the king is in check and there are no possible moves for the player to make, it is in checkmate	Yes Test: 4.5.3	This test proves that the game correctly identifies a checkmate which fulfils the requirement.
5.2.2	If a king is in checkmate, the user should be notified, and the game should end	Yes Test: 4.5.3	This test proves that the game correctly handles a checkmate which fulfils the requirement.
5.3.1	If a king is not threatened by an opponent's piece, but there is no possible moves to make, this is a stalemate	Yes Test: 4.5.4	This test proves that the game correctly identifies a stalemate which fulfils the requirement
5.3.2	The user should be notified, and the game should end (in a draw)	Yes Test: 4.5.4	This test proves the game correctly handles a stalemate by ending the game and notifying the user which fulfils the requirement.
6.1	The game must be automatically saved after every move	Yes Test: 4.6.4	This test proves that the game is loaded and saved correctly and automatically at the end of each move. This fulfils the requirement.

6.1.1	The board state must be saved using a specific format for it to be read again by the program	Yes Tests: 5.1.6, 4.6.1	4.6.1 proves the board save is accurate by comparing the loaded version to the saved version. 5.1.6 proves the board save is accurate in a beta testing scenario. This fulfils the requirement.
6.1.2	Whoever's turn it is next must be saved	Yes Test: 4.6.2	This test proves the player who's turn it is next is being saved correctly in the board save file. This fulfils the requirement.
6.2	A GUI must be used to ask (and get a response from) the user if they want to load the previous game or start fresh	Yes Tests: 5.1.6,4.6.3	4.6.3 proves there is a prompt displayed at the start of the game asking to load or not. 5.1.6 proves the prompt is simple and easy to use in a beta testing scenario. These test fulfil the requirement.

All input an outputs for menus and files are tested in the IO testing section (1).

Improvements

The move piece functionality is not the smoothest as the start square and end square must be selected individually. This could be improved by implementing dragging pieces and/or movement animations to make the game a more immersive, realistic, and enjoyable experience. This could be done by sensing when the click is held over a piece, and moving the piece along with the cursor until the click is released at which point the end square is identified.

The loading bar system that is in place for the highest level of competency displays in the text output of the program. This is difficult to read and is not obvious. This can be improved by moving it to a graphical representation on the UI. This would make the wait time for the AI move generation more bearable since you can see roughly how long is left. This can be done by creating a new menu style UI window which will be shown whenever the AI is calculating and will use similar code to the text based current one.

Sometimes it's not obvious who's turn the game is on. It would be useful to add a box on the UI that shows who's turn it currently is so you're not trying to make moves when it's not your turn, and you are not waiting for no reason when it is your turn. This would be done by adding a box that can be updated with the current players turn whenever the rest of the board is also updated.

The minmax is only so difficult to beat, to make it more challenging for experienced players the depth could be increased. This would cause the computer to take much longer to calculate the moves. Therefore, more optimisation to the minmax algorithm could be added to allow for the higher and more difficult to beat depths. AN example of an improvement is paralysation (having multiple processes at once) which can be achieved using the python multiprocessing library.

The minmax currently runs at a fixed depth of 2. The user could be able to increase this depth to higher levels to make the game more challenging if they desire. This could be don't using a more advanced difficulty selection menu at the start of the game which includes some kind of user input for the depth of the AI when you select the hard difficulty level.

Possible Additional Features

There could be an option to export the chess notation for a game as a file. This would enable people to save previous games as records and possibly transfer them to other chess playing engines. This would be done by recording the chess notation live in a game and constantly appending it to an exportable text file.

A feature to restart the game once its ended could be added. This would occur when any game ends and could be a button that would quickly reset and restart the game. A separate method in the board object would have to be added in order to fully reset the AI and the board.

A forfeit button could be added to give the player the option to give up if they don't want to continue. This would help not having to end and run the program again to restart. This could be done by adding a button in one of the corners of the UI which triggers an end game sequence. This would be done as well as the game restart feature.

A player leader board could be added based on players achievement against the AI. This would give players a target and allow them to compare to other players. This would be accessible through a button that could be added to one of the corners and would be done through a database that could be accessible through the internet. It would be an additional window that shows the top players and your current top score.

A personal high score system could be added to give users a target to aim for next time they play. This would be a number that saves in a file that would be displayed to the user in one of the borders of the UI. The number should be automatically updated when a higher score is achieved.

A chess clock could be added which would be a individual countdown for each player that only counts down when it's that players turn (the player has a max time limit for the whole game). Once this countdown reaches zero for a player, that player loses. This would make the game more realistic to competitive chess as chess clocks are a very common feature in competitions. It could be done by storing the countdown for each player in separate variables. The countdowns could then be handled on a separate threaded process that runs alongside the AI. The global variable can then be changed to indicate when the timer has run out or who’s turn it is.

Technical Solution

Quick example video: https://www.youtube.com/watch?v=bt6OdBQQYig

Code

```
# IMPORTANT: Image files of chess pieces are required to be in a pieces folder in
the same directory as the python script

import copy, abc, pygame, random, time

DEPTH = 2
FILE_NAME_LOAD = 'board.txt'
FILE_NAME_SAVE = 'board.txt'

ANSI_codes = {'RED':'\033[91m', 'GREEN':'\033[92m', 'YELLOW':'\033[93m',
'BLUE':'\033[94m', 'MAGENTA':'\033[95m', 'CYAN':'\033[96m', 'WHITE':'\033[97m',
'END':'\033[0m'}

class UserInterface():
    def __init__(self):
        LIGHT_COLOUR = (246, 213, 180)
        DARK_COLOUR = (202, 157, 111)
        EDGE_COLOUR = (0,0,0)

        pygame.init()
        pygame.font.init()

        self.__screen = pygame.display.set_mode((680,680))
        pygame.display.set_caption("AI Chess Game")

        margin = pygame.Surface((680,680))
        margin.fill(DARK_COLOUR)
        self.__screen.blit(margin, (0, 0))

        margin2 = pygame.Surface((602,602))
        margin2.fill(EDGE_COLOUR)
        self.__screen.blit(margin2, (39, 39))


        font = pygame.font.SysFont('arialblack.ttf', 32)
        for i in range(8):
            letter = font.render(chr(i+65),True,EDGE_COLOUR)
            number = font.render(str(8-i),True,EDGE_COLOUR)
            self.__screen.blit(letter, (i*75+70, 10))
            self.__screen.blit(number, (15, i*75+65))

        board = pygame.Surface((600, 600))
```

```
        for x in range(0,8,2):
            for y in range(0,8,2):
                pygame.draw.rect(board, LIGHT_COLOUR, (x*75, y*75, 75, 75))
        for x in range(1,8,2):
            for y in range(1,8,2):
                pygame.draw.rect(board, LIGHT_COLOUR, (x*75, y*75, 75, 75))

        for x in range(1,8,2):
            for y in range(0,8,2):
                pygame.draw.rect(board, DARK_COLOUR, (x*75, y*75, 75, 75))
        for x in range(0,8,2):
            for y in range(1,8,2):
                pygame.draw.rect(board, DARK_COLOUR, (x*75, y*75, 75, 75))
        self.__screen.blit(board, (40, 40))

        pygame.display.update()

    def update_board(self,board_to_use,green_highlight=[],red_highlight=[]):
        LIGHT_COLOUR = (246, 213, 180)
        DARK_COLOUR = (202, 157, 111)
        EDGE_COLOUR = (0,0,0)

        margin = pygame.Surface((680,680))
        margin.fill(DARK_COLOUR)
        self.__screen.blit(margin, (0, 0))

        margin2 = pygame.Surface((602,602))
        margin2.fill(EDGE_COLOUR)
        self.__screen.blit(margin2, (39, 39))


        font = pygame.font.SysFont('arialblack.ttf', 32)
        for i in range(8):
            letter = font.render(chr(i+65),True,EDGE_COLOUR)
            number = font.render(str(8-i),True,EDGE_COLOUR)
            self.__screen.blit(letter, (i*75+70, 10))
            self.__screen.blit(number, (15, i*75+65))

        board = pygame.Surface((600, 600))

        for x in range(0,8,2):
            for y in range(0,8,2):
                pygame.draw.rect(board, LIGHT_COLOUR, (x*75, y*75, 75, 75))
        for x in range(1,8,2):
            for y in range(1,8,2):
                pygame.draw.rect(board, LIGHT_COLOUR, (x*75, y*75, 75, 75))
```

```
        for x in range(1,8,2):
            for y in range(0,8,2):
                pygame.draw.rect(board, DARK_COLOUR, (x*75, y*75, 75, 75))
        for x in range(0,8,2):
            for y in range(1,8,2):
                pygame.draw.rect(board, DARK_COLOUR, (x*75, y*75, 75, 75))

        self.__screen.blit(board, (40, 40))

        for square in green_highlight:
            x = square[0]
            y = square[1]
            green_square = pygame.Surface((75,75),pygame.SRCALPHA,32)
            green_square = green_square.convert_alpha()
            green_square.fill((0,200,0,50))
            self.__screen.blit(green_square, (75*x + 40, 75*y + 40))

        for square in red_highlight:
            x = square[0]
            y = square[1]
            green_square = pygame.Surface((75,75),pygame.SRCALPHA,32)
            green_square = green_square.convert_alpha()
            green_square.fill((255,0,0,50))
            self.__screen.blit(green_square, (75*x + 40, 75*y + 40))

        for row in range(8):
            for column in range(8):
                square = board_to_use[row][column]
                if square != None:
                    t = square.get_type().lower()
                    c = square.get_colour().lower()
                    image_obj = pygame.image.load(f'pieces/{t}_{c}.webp')
                    image_obj = pygame.transform.scale(image_obj,(80,80))
                    self.__screen.blit(image_obj, (75*column+37, 75*row+35))

        pygame.display.update()

    def show_dificulty(self):
        grey_background = pygame.Surface((680,680),pygame.SRCALPHA,32)
        grey_background = grey_background.convert_alpha()
        grey_background.fill((0, 0, 0,150))
        self.__screen.blit(grey_background, (0, 0))

        difficulty_window = pygame.Surface((400,180))
        difficulty_window.fill((246, 213, 180))
        self.__screen.blit(difficulty_window, (140, 250))

        font = pygame.font.SysFont('arialblack.ttf', 32)
```

```
        difficulty_title = font.render('Choose a difficulty level:',True,(0,0,0))
        self.__screen.blit(difficulty_title,(150,260))

        low_difficulty = pygame.Surface((100,50))
        low_difficulty.fill((202, 157, 111))
        self.__screen.blit(low_difficulty, (180, 330))
        low_difficulty_title = font.render('Easy',True,(0,0,0))
        self.__screen.blit(low_difficulty_title,(205,345))

        mid_difficulty = pygame.Surface((100,50))
        mid_difficulty.fill((202, 157, 111))
        self.__screen.blit(mid_difficulty, (290, 330))
        mid_difficulty_title = font.render('Med',True,(0,0,0))
        self.__screen.blit(mid_difficulty_title,(320,345))

        high_difficulty = pygame.Surface((100,50))
        high_difficulty.fill((202, 157, 111))
        self.__screen.blit(high_difficulty, (400, 330))
        high_difficulty_title = font.render('Hard',True,(0,0,0))
        self.__screen.blit(high_difficulty_title,(425,345))

        pygame.display.update()

    def show_promotion(self,colour):
        grey_background = pygame.Surface((680,680),pygame.SRCALPHA,32)
        grey_background = grey_background.convert_alpha()
        grey_background.fill((0, 0, 0,150))
        self.__screen.blit(grey_background, (0, 0))

        window = pygame.Surface((400,180))
        window.fill((246, 213, 180))
        self.__screen.blit(window, (140, 250))

        font = pygame.font.SysFont('arialblack.ttf', 32)
        title = font.render('Choose a piece to promote to:',True,(0,0,0))
        self.__screen.blit(title,(150,260))

        knight = pygame.Surface((80,80))
        knight.fill((202, 157, 111))
        self.__screen.blit(knight, (200, 330))
        knight_image =
pygame.transform.scale(pygame.image.load(f'pieces/n_{colour.lower()}.webp'),(80
,80))
        self.__screen.blit(knight_image,(200, 330))

        queen = pygame.Surface((80,80))
        queen.fill((202, 157, 111))
        self.__screen.blit(queen, (400, 330))
```

```
        queen_image =
pygame.transform.scale(pygame.image.load(f'pieces/q_{colour.lower()}.webp'),(80
,80))
        self.__screen.blit(queen_image,(400,330))

        pygame.display.update()

    def show_game_end(self,colour=None):
        grey_background = pygame.Surface((680,680),pygame.SRCALPHA,32)
        grey_background = grey_background.convert_alpha()
        grey_background.fill((0, 0, 0,150))
        self.__screen.blit(grey_background, (0, 0))

        font = pygame.font.SysFont('arialblack.ttf', 32)
        if colour == 'B':
            window = pygame.Surface((500,60))
            window.fill((246, 213, 180))
            self.__screen.blit(window, (90, 310))
            end_game_title = font.render('Checkmate against Black, White
Wins!',True,(0,0,0))
            self.__screen.blit(end_game_title,(130,327))
        elif colour == 'W':
            window = pygame.Surface((500,60))
            window.fill((246, 213, 180))
            self.__screen.blit(window, (90, 310))
            end_game_title = font.render('Checkmate against White. Black
Wins!',True,(0,0,0))
            self.__screen.blit(end_game_title,(130,327))
        else:
            window = pygame.Surface((200,60))
            window.fill((246, 213, 180))
            self.__screen.blit(window, (240, 310))
            end_game_title = font.render('Stalemate, Draw!',True,(0,0,0))
            self.__screen.blit(end_game_title,(250,327))

        pygame.display.update()

    def load_menu(self):
        grey_background = pygame.Surface((680,680),pygame.SRCALPHA,32)
        grey_background = grey_background.convert_alpha()
        grey_background.fill((0, 0, 0,150))
        self.__screen.blit(grey_background, (0, 0))

        window = pygame.Surface((400,180))
        window.fill((246, 213, 180))
        self.__screen.blit(window, (140, 250))

        font = pygame.font.SysFont('arialblack.ttf', 32)
```

```
        title = font.render('Load previous game?',True,(0,0,0))
        self.__screen.blit(title,(150,260))

        yes_btn = pygame.Surface((120,50))
        yes_btn.fill((202, 157, 111))
        self.__screen.blit(yes_btn, (190, 335))
        yes_title = font.render('Yes',True,(0,0,0))
        self.__screen.blit(yes_title,(230,350))

        no_btn = pygame.Surface((120,50))
        no_btn.fill((202, 157, 111))
        self.__screen.blit(no_btn, (370, 335))
        no_title = font.render('No',True,(0,0,0))
        self.__screen.blit(no_title,(418,350))

        pygame.display.update()

class Game():
    def __init__(self):
        self.__game_end = False

        UI = UserInterface()
        board = Board()
        UI.load_menu()
        self.__loading_from_file = False
        self.__difficulty_open = False
        self.__game_started = False
        self.__promotion_open = False
        self.__piece_to_promote = None
        self.__exit_game = False
        self.__load_open = True

        self.__difficulty = -1
        turn = 'W' # White goes first
        coords_old = None
        coords_new = None
        pressed = False

        while not self.__exit_game:

            for event in pygame.event.get():
                if event.type == pygame.QUIT:
                    pygame.quit()

            if self.__load_open:
                if pygame.mouse.get_pressed()[0] and not pressed:
                    pressed = True
                    x_pos = pygame.mouse.get_pos()[0]
```

```
                y_pos = pygame.mouse.get_pos()[1]
                if x_pos > 190 and x_pos < 310 and y_pos > 335 and y_pos < 385:
                    self.__loading_from_file = True
                    UI.update_board(board.get_board())
                    self.__load_open = False
                    UI.show_dificulty()
                    self.__difficulty_open = True
                elif x_pos > 370 and x_pos < 490 and y_pos > 335 and y_pos < 385:
                    self.__loading_from_file = False
                    UI.update_board(board.get_board())
                    self.__load_open = False
                    UI.show_dificulty()
                    self.__difficulty_open = True

        elif self.__difficulty_open: # Difficulty not selected
            if pygame.mouse.get_pressed()[0] and not pressed:
                pressed = True
                x_pos = pygame.mouse.get_pos()[0]
                y_pos = pygame.mouse.get_pos()[1]
                if x_pos > 180 and x_pos < 280 and y_pos > 330 and y_pos < 380:
                    self.__difficulty = 0
                    UI.update_board(board.get_board())
                    self.__difficulty_open = False
                elif x_pos > 290 and x_pos < 390 and y_pos > 330 and y_pos < 380:
                    self.__difficulty = 1
                    UI.update_board(board.get_board())
                    self.__difficulty_open = False
                elif x_pos > 400 and x_pos < 500 and y_pos > 330 and y_pos < 380:
                    self.__difficulty = 2
                    UI.update_board(board.get_board())
                    self.__difficulty_open = False

        elif not self.__game_started: # Game not started but needs to start
            player1 = User('W')
            player2 = Computer('B',self.__difficulty,depth=DEPTH)
            self.__game_started = True
            if self.__loading_from_file:
                turn = board.load_board(FILE_NAME_LOAD)
                UI.update_board(board.get_board())

        elif self.__promotion_open:
            if pygame.mouse.get_pressed()[0] and not pressed:
                pressed = True
                x_pos = pygame.mouse.get_pos()[0]
                y_pos = pygame.mouse.get_pos()[1]
                if x_pos > 200 and x_pos < 280 and y_pos > 330 and y_pos < 410:
                    self.__piece_to_promote.promote('N')
                    UI.update_board(board.get_board())
```

```
                    self.__promotion_open = False

                elif x_pos > 400 and x_pos < 480 and y_pos > 330 and y_pos < 410:
                    self.__piece_to_promote.promote('Q')
                    UI.update_board(board.get_board())
                    self.__promotion_open = False

                if self.__promotion_open == False:
                    if board.valid_moves(turn, check=True) == []:
                        king = None
                        for row in board.get_board():
                            for square in row:
                                if square != None:
                                    if square.get_type() == 'K' and square.get_colour() ==
turn:
                                        king = square
                                        break
                        if board.check_for_check(king):
                            self.__end_game(turn)
                            UI.show_game_end(turn)
                        else:
                            self.__end_game()
                            UI.show_game_end()

        elif not self.__game_end: # Game Started
            player_turn = player1
            if player1.get_colour() == turn:
                player_turn = player1
            else:
                player_turn = player2

            if pygame.mouse.get_pressed()[0] and not pressed and
player_turn.get_type() == 'User':

                pressed = True
                x_pos = pygame.mouse.get_pos()[0]
                y_pos = pygame.mouse.get_pos()[1]
                if x_pos > 40 and x_pos < 640 and y_pos > 40 and y_pos < 640:
                    x = ((x_pos-40) // 75)
                    y = ((y_pos-40) // 75)
                    square = board.get_board()[y][x]
                    if square != None:
                        colour = square.get_colour()
                        if colour.upper() == turn and coords_old == None:
                            coords_old = board.calculate_coordinates(x=x,y=y)
                            #print(f'Playing: {coords_old}')
                            UI.update_board(board.get_board(),green_highlight=[[x,y]])
                        elif colour.upper() == turn and coords_old != None:
```

```
                                x_old,y_old = board.calculate_coordinates(coord=coords_old)
                                if board.get_board()[y][x].get_type() == 'K' and
board.get_board()[y_old][x_old].get_type() == 'R':
                                    if board.move_piece(coords_old,None,castle=True):
                                        print('Castling')
                                        board.check_pos()
                                        coords_old = None
                                        coords_new = None

UI.update_board(board.get_board(),red_highlight=[[x_old,y_old],[x,y]])
                                        if turn == 'B':
                                            turn = 'W'
                                        else:
                                            turn = 'B'
                                        if board.valid_moves(turn, check=True) == []:
                                            king = None
                                            for row in board.get_board():
                                                for square in row:
                                                    if square != None:
                                                        if square.get_type() == 'K' and
square.get_colour() == turn:
                                                            king = square
                                                            break
                                            if board.check_for_check(king):
                                                self.__end_game(turn)
                                                UI.show_game_end(turn)
                                            else:
                                                self.__end_game()
                                                UI.show_game_end()
                                        else:

board.save_board(name=FILE_NAME_SAVE,turn=turn)

                                    else:
                                        coords_old = None
                                        coords_new = None
                                        UI.update_board(board.get_board())
                                else:
                                    coords_old = None
                                    coords_new = None
                                    UI.update_board(board.get_board())

                        if coords_new == None and coords_old != None:
                                if square != None:
                                    if square.get_colour() != turn:
                                        coords_new = board.calculate_coordinates(x=x,y=y)
                                        #print(f'To: {coords_new}')
                                        if board.move_piece(coords_old,coords_new):
```

```
                                    board.check_pos()
                                    x_old,y_old =
board.calculate_coordinates(coord=coords_old)
                                    coords_old = None
                                    coords_new = None

UI.update_board(board.get_board(),red_highlight=[[x_old,y_old],[x,y]])
                                    if board.promotion_check(x,y)[0]:
                                        UI.show_promotion(turn)
                                        self.__piece_to_promote =
board.promotion_check(x,y)[1]
                                        self.__promotion_open = True
                                    if turn == 'B':
                                        turn = 'W'
                                    else:
                                        turn = 'B'
                                    if board.valid_moves(turn, check=True) == []:
                                        king = None
                                        for row in board.get_board():
                                            for square in row:
                                                if square != None:
                                                    if square.get_type() == 'K' and
square.get_colour() == turn:
                                                        king = square
                                                        break
                                        if board.check_for_check(king):
                                            self.__end_game(turn)
                                            UI.show_game_end(turn)
                                        else:
                                            self.__end_game()
                                            UI.show_game_end()
                                    else:

board.save_board(name=FILE_NAME_SAVE,turn=turn)

                                else:
                                    coords_old = None
                                    coords_new = None
                                    UI.update_board(board.get_board())
                        else:
                            coords_new = board.calculate_coordinates(x=x,y=y)
                            #print(f'To: {coords_new}')
                            if board.move_piece(coords_old,coords_new):
                                board.check_pos()
                                x_old,y_old =
board.calculate_coordinates(coord=coords_old)
                                coords_old = None
                                coords_new = None
```

```
UI.update_board(board.get_board(),red_highlight=[[x_old,y_old],[x,y]])
                              if board.promotion_check(x,y)[0]:
                                 UI.show_promotion(turn)
                                 self.__piece_to_promote =
board.promotion_check(x,y)[1]
                                 self.__promotion_open = True
                              if turn == 'B':
                                 turn = 'W'
                              else:
                                 turn = 'B'
                              if board.valid_moves(turn, check=True) == []:
                                 king = None
                                 for row in board.get_board():
                                    for square in row:
                                       if square != None:
                                          if square.get_type() == 'K' and
square.get_colour() == turn:
                                             king = square
                                             break
                                 if board.check_for_check(king):
                                    self.__end_game(turn)
                                    UI.show_game_end(turn)
                                 else:
                                    self.__end_game()
                                    UI.show_game_end()
                              else:

board.save_board(name=FILE_NAME_SAVE,turn=turn)

                           else:
                              coords_old = None
                              coords_new = None
                              UI.update_board(board.get_board())

            elif player_turn.get_type() == 'Computer':
               done = False
               x_old,y_old,x_new,y_new =
player_turn.get_move(board.get_board(),board)
               start_square = board.get_board()[y_old][x_old]
               try:
                  end_square = board.get_board()[y_new][x_new]
               except:
                  end_square = None
               try:
                  if end_square == None and start_square.get_type() == 'R':
```

```
                    if
board.move_piece(board.calculate_coordinates(x=x_old,y=y_old),None,castle=Tru
e):
                        board.check_pos()

UI.update_board(board.get_board(),red_highlight=[[x_old,y_old],[4,y_old]])
                        if turn == 'B':
                            turn = 'W'
                        else:
                            turn = 'B'
                        if board.valid_moves(turn, check=True) == []:
                            king = None
                            for row in board.get_board():
                                for square in row:
                                    if square != None:
                                        if square.get_type() == 'K' and square.get_colour()
== turn:
                                            king = square
                                            break
                            if board.check_for_check(king):
                                self.__end_game(turn)
                                UI.show_game_end(turn)
                            else:
                                self.__end_game()
                                UI.show_game_end()
                        else:
                            board.save_board(name=FILE_NAME_SAVE,turn=turn)
                        done = True
            except: pass

            if not done:
                if
board.move_piece(board.calculate_coordinates(x=x_old,y=y_old),board.calculate_
coordinates(x=x_new,y=y_new)):
                    board.check_pos()
                    if board.promotion_check(x_new,y_new)[0]:
                        board.promotion_check(x_new,y_new)[1].promote('Q')

UI.update_board(board.get_board(),red_highlight=[[x_old,y_old],[x_new,y_new]])
                    if turn == 'B':
                        turn = 'W'
                    else:
                        turn = 'B'
                    if board.valid_moves(turn, check=True) == []:
                        king = None
                        for row in board.get_board():
                            for square in row:
                                if square != None:
```

```
                                            if square.get_type() == 'K' and square.get_colour() ==
turn:
                                                king = square
                                                break
                                if board.check_for_check(king):
                                    self.__end_game(turn)
                                    UI.show_game_end(turn)
                                else:
                                    self.__end_game()
                                    UI.show_game_end()
                            else:
                                board.save_board(name=FILE_NAME_SAVE,turn=turn)

            elif self.__game_end:
                continue

            if not pygame.mouse.get_pressed()[0] and pressed:
                pressed = False

    def __end_game(self, colour=None):
        self.__game_end = True
        if colour == 'B':
            print('Black has lost, WHITE WINS')
        elif colour == 'W':
            print('White has lost. BLACK WINS')
        else:
            print('Game Over. DRAW')

class Board():
    def __init__(self):
        self.__setup_board()

    def display_board(self, board_to_use=None):
        if board_to_use == None:
            board_to_use = self.__board_array

        pieces = {'BP':'♟', 'WP':'♙', 'BR':'♜', 'WR':'♖', 'BN':'♞', 'WN':'♘', 'BB':'♝',
'WB':'♗', 'BQ':'♛', 'WQ':'♕', 'BK':'♚', 'WK':'♔'}
        print('  a  b  c  d  e  f  g  h')
        for row in range(8):
            print(8-row, end='')
            for column in range(8):
                square = board_to_use[row][column]
                if square != None:
                    search_string = f'{square.get_colour()}{square.get_type()}'
                    print(f'[{pieces[search_string]}]',end='')
                else:
                    print('[ ]',end='')
```

```
            print()

    def __setup_board(self):
        self.__board_array = [
            [Piece('R','a8', 'B'),Piece('N','b8', 'B'),Piece('B','c8', 'B'),Piece('Q','d8',
'B'),Piece('K','e8', 'B'),Piece('B','f8', 'B'),Piece('N','g8', 'B'),Piece('R','h8', 'B')],
            [Piece('P','a7', 'B'),Piece('P','b7', 'B'),Piece('P','c7', 'B'),Piece('P','d7',
'B'),Piece('P','e7', 'B'),Piece('P','f7', 'B'),Piece('P','g7', 'B'),Piece('P','h7', 'B')],
            [None,None,None,None,None,None,None,None],
            [None,None,None,None,None,None,None,None],
            [None,None,None,None,None,None,None,None],
            [None,None,None,None,None,None,None,None],
            [Piece('P','a2', 'W'),Piece('P','b2', 'W'),Piece('P','c2', 'W'),Piece('P','d2',
'W'),Piece('P','e2', 'W'),Piece('P','f2', 'W'),Piece('P','g2', 'W'),Piece('P','h2', 'W')],
            [Piece('R','a1', 'W'),Piece('N','b1', 'W'),Piece('B','c1', 'W'),Piece('Q','d1',
'W'),Piece('K','e1', 'W'),Piece('B','f1', 'W'),Piece('N','g1', 'W'),Piece('R','h1', 'W')]
        ]

    def save_board(self, board=None, turn='W', name='board.txt'):
        if board == None:
            board = self.__board_array

        final_save = f'{turn} '

        for row in board:
            for square in row:
                if square == None:
                    final_save = f'{final_save}[None] '
                else:
                    final_save = f'{final_save}[{square.get_type()} {square.get_location()}
{square.get_colour()}] '

        file = open(name,'w')
        file.write(final_save)
        file.close()

    def load_board(self, name='board.txt'): # Loads the board into place and returns
whos turn your on
        file = open(name,'r')
        text = file.read()
        file.close()

        if text == '':
            print('The file is empty.')
            exit()

        new_board = [
            [None,None,None,None,None,None,None,None],
```

```
            [None,None,None,None,None,None,None,None],
            [None,None,None,None,None,None,None,None],
            [None,None,None,None,None,None,None,None],
            [None,None,None,None,None,None,None,None],
            [None,None,None,None,None,None,None,None],
            [None,None,None,None,None,None,None,None],
            [None,None,None,None,None,None,None,None]
        ]

        turn = text[:text.find(' ')]
        text = text[text.find(' ')+1:]

        if text == '':
            print('The file is empty.')
            exit()

        while text != '':
            object_str = text[text.find('[')+1:text.find(']')]
            text = text[text.find(']')+2:]

            if object_str == 'None':
                continue
            else:
                piece_type = object_str[:object_str.find(' ')]
                object_str = object_str[object_str.find(' ')+1:]
                piece_loc = object_str[:object_str.find(' ')]
                object_str = object_str[object_str.find(' ')+1:]
                piece_colour = object_str

                if not (piece_colour == 'W' or piece_colour == 'B'):
                    print("The colour of a piece is invalid.")
                    exit()

                if not (piece_type == 'P' or piece_type == 'R' or piece_type == 'N' or
piece_type == 'B' or piece_type == 'Q' or piece_type == 'K'):
                    print("The type of a piece is invalid.")
                    exit()

                try:
                    x,y = self.calculate_coordinates(coord=piece_loc)
                except:
                    print("The coordinates of a piece are invalid.")
                    exit()

                try:
                    new_board[y][x] = Piece(piece_type,piece_loc,piece_colour)
                except:
                    print('There was an error creating a piece.')
```

```
                exit()

        self.__board_array = copy.deepcopy(new_board)
        return turn

    def calculate_coordinates(self,coord=None,x=None,y=None):
        if coord != None:
            x = coord[0]
            y = int(coord[1])
            return (ord(x)-97), 7-(y-1)
        elif x != None and y != None:
            return f'{(chr(x+97))}{7-(y-1)}'

    def move_piece(self, coord_old, coord_new, castle=False):
        x_old, y_old = self.calculate_coordinates(coord_old)
        if not castle:
            x_new, y_new = self.calculate_coordinates(coord_new)
        else:
            x_new = None
            y_new = None

        piece_type = self.__board_array[y_old][x_old].get_type()
        piece_colour = self.__board_array[y_old][x_old].get_colour()

        validation = self.validate_move(x_old,y_old,x_new,y_new,castle)

        if validation and castle:
            if x_old == 0:
                self.__board_array[y_old][3] = self.__board_array[y_old][x_old]
                self.__board_array[y_old][3].change_location(f'd{abs(y_old-8)}')
                self.__board_array[y_old][x_old] = None
                self.__board_array[y_old][2] = self.__board_array[y_old][4]
                self.__board_array[y_old][2].change_location(f'c{abs(y_old-8)}')
                self.__board_array[y_old][4] = None

                self.__board_array[y_old][3].register_moved()
                self.__board_array[y_old][2].register_moved()
            elif x_old == 7:
                self.__board_array[y_old][5] = self.__board_array[y_old][x_old]
                self.__board_array[y_old][5].change_location(f'f{abs(y_old-8)}')
                self.__board_array[y_old][x_old] = None
                self.__board_array[y_old][6] = self.__board_array[y_old][4]
                self.__board_array[y_old][6].change_location(f'g{abs(y_old-8)}')
                self.__board_array[y_old][4] = None

                self.__board_array[y_old][5].register_moved()
                self.__board_array[y_old][6].register_moved()
```

```
            for row in self.__board_array:
                for piece in row:
                    if piece != None:
                        piece.amend_en_passant(False)
            return True

        elif validation == True:

            self.__board_array[y_new][x_new] = self.__board_array[y_old][x_old]
            self.__board_array[y_new][x_new].change_location(coord_new)
            self.__board_array[y_old][x_old] = None

            self.__board_array[y_new][x_new].register_moved()

            for row in self.__board_array:
                for piece in row:
                    if piece != None:
                        piece.amend_en_passant(False)

            if piece_type == 'P' and abs(y_new-y_old) == 2:
                self.__board_array[y_new][x_new].amend_en_passant(True)

            return True

        elif validation == (True,True):

            self.__board_array[y_new][x_new] = self.__board_array[y_old][x_old]
            self.__board_array[y_new][x_new].change_location(coord_new)
            self.__board_array[y_old][x_old] = None
            self.__board_array[y_old][x_new] = None

            self.__board_array[y_new][x_new].register_moved()

            for row in self.__board_array:
                for piece in row:
                    if piece != None:
                        piece.amend_en_passant(False)

            return True

        else:
            return False

    def validate_move(self, x_old, y_old, x_new, y_new, castle=False,
board=None):

        if board == None:
            board = self.__board_array
```

```
        if board[y_old][x_old] != None:
            piece_type = board[y_old][x_old].get_type()
            piece_colour = board[y_old][x_old].get_colour()

        coord_new = self.calculate_coordinates(x=x_new,y=y_new)

        if castle and board[y_old][4] == None:
            return False
        elif castle:
            if board[y_old][4].get_type() != 'K' or board[y_old][4].get_colour() !=
piece_colour:
                return False

        new_board = copy.deepcopy(board)
        if not castle:
            new_board[y_new][x_new] = new_board[y_old][x_old]
            new_board[y_old][x_old] = None
            new_board[y_new][x_new].change_location(coord_new)
        elif x_old == 0:
            new_board[y_old][3] = new_board[y_old][x_old]
            new_board[y_old][3].change_location(f'd{abs(y_old-8)}')
            new_board[y_old][x_old] = None
            new_board[y_old][2] = new_board[y_old][4]
            new_board[y_old][2].change_location(f'c{abs(y_old-8)}')
            new_board[y_old][4] = None
        else:
            new_board[y_old][5] = new_board[y_old][x_old]
            new_board[y_old][5].change_location(f'f{abs(y_old-8)}')
            new_board[y_old][x_old] = None
            new_board[y_old][6] = new_board[y_old][4]
            new_board[y_old][6].change_location(f'g{abs(y_old-8)}')
            new_board[y_old][4] = None

        new_king = False
        for row in new_board:
            for pieceCol in row:
                if pieceCol != None:
                    if pieceCol.get_type() == 'K' and pieceCol.get_colour() ==
piece_colour:
                        new_king = pieceCol
                        break
            if new_king != False:
                break

        king = False
        for row in board:
            for pieceCol in row:
```

```
            if pieceCol != None:
                if pieceCol.get_type() == 'K' and pieceCol.get_colour() ==
piece_colour:
                    king = pieceCol
                    break
        if king != False:
            break
    if self.check_for_check(new_king, new_board):
        return False

    # SEPERATE VALIDATION FOR CASTLING
    if castle:
        if self.check_for_check(king, board):
            return False
        piece = board[y_old][x_old]
        if piece_type != 'R' or piece.get_moved():
            return False

        king_slot = board[y_old][4]
        if king_slot != None:
            if king_slot.get_type() == 'K' and not king_slot.get_moved():
                if x_old == 0:
                    for i in range(1,4):
                        if board[y_old][i] != None:
                            return False
                    return True
                elif x_old == 7:
                    for i in range(5,7):
                        if board[y_old][i] != None:
                            return False
                    return True
        return False

    end_square_entity = board[y_new][x_new]

    if board[y_new][x_new] != None:
        if board[y_new][x_new].get_colour().upper() == piece_colour:
            return False

    if x_new < 0 or x_new > 7 or y_new < 0 or y_new > 7:
        return False

    if y_old == y_new and x_old == x_new:
        return False

    if piece_type == 'P' and piece_colour == 'W': # White Pawn Validation
        if x_old == x_new and y_old == y_new + 1 and end_square_entity ==
None:
```

```
                return True
            elif (x_old == x_new - 1 or x_old == x_new + 1) and y_old == y_new + 1
and end_square_entity != None:
                return True
            elif x_old == x_new and y_old == y_new + 2 and end_square_entity ==
None and y_old == 6 and board[y_old-1][x_old] == None:
                return True
            # EN PASSANT:
            elif (x_old == x_new + 1 or x_old == x_new - 1) and y_old == y_new + 1
and end_square_entity == None and self.__board_array[y_old][x_new] != None:
                if board[y_old][x_new].get_colour() != piece_colour and
self.__board_array[y_old][x_new].get_en_passant() == True and
self.__board_array[y_old][x_new].get_type() == 'P':
                    return True, True

        elif piece_type == 'P' and piece_colour == 'B': # Black Pawn Validation
            if x_old == x_new and y_old == y_new - 1 and end_square_entity == None:
                return True
            elif (x_old == x_new - 1 or x_old == x_new + 1) and y_old == y_new - 1
and end_square_entity != None:
                return True
            elif x_old == x_new and y_old == y_new - 2 and end_square_entity ==
None and y_old == 1  and board[y_old+1][x_old] == None:
                return True
            # EN PASSANT:
            elif (x_old == x_new + 1 or x_old == x_new - 1) and y_old == y_new - 1
and end_square_entity == None and self.__board_array[y_old][x_new] != None:
                if board[y_old][x_new].get_colour() != piece_colour and
self.__board_array[y_old][x_new].get_en_passant() == True and
self.__board_array[y_old][x_new].get_type() == 'P':
                    return True, True

        elif piece_type == 'R': # All Rook Validation
            if x_old == x_new and y_old < y_new:
                blocker = False
                for i in range(y_old+1,y_new):
                    if board[i][x_old] != None:
                        blocker = True
                if not blocker:
                    return True

            elif x_old == x_new and y_old > y_new:
                blocker = False
                for i in range(y_new+1,y_old):
                    if board[i][x_old] != None:
                        blocker = True
                if not blocker:
                    return True
```

```
    elif y_old == y_new and x_old < x_new:
        blocker = False
        for i in range(x_old+1,x_new):
            if board[y_old][i] != None:
                blocker = True
        if not blocker:
            return True

    elif y_old == y_new and x_old > x_new:
        blocker = False
        for i in range(x_new+1,x_old):
            if board[y_old][i] != None:
                blocker = True
        if not blocker:
            return True

elif piece_type == 'N': # All Knight Validation
    if x_old == x_new-1 and y_old == y_new+2:
        return True
    elif x_old == x_new-2 and y_old == y_new+1:
        return True
    elif x_old == x_new-2 and y_old == y_new-1:
        return True
    elif x_old == x_new-1 and y_old == y_new-2:
        return True
    elif x_old == x_new+1 and y_old == y_new-2:
        return True
    elif x_old == x_new+2 and y_old == y_new-1:
        return True
    elif x_old == x_new+2 and y_old == y_new+1:
        return True
    elif x_old == x_new+1 and y_old == y_new+2:
        return True

elif piece_type == 'B': # All Bishop Validation
    if x_old - x_new == y_old - y_new:
        blocker = False
        if x_old > x_new:
            for i in range(1,x_old-x_new):
                if board[y_old-i][x_old-i] != None:
                    blocker = True
        else:
            for i in range(1,x_new-x_old):
                if board[y_old+i][x_old+i] != None:
                    blocker = True
        if not blocker:
            return True
```

```
        elif x_old - x_new == -(y_old - y_new):
            blocker = False
            if x_old > x_new:
                for i in range(1,x_old-x_new):
                    if board[y_old+i][x_old-i] != None:
                        blocker = True
            else:
                for i in range(1,x_new-x_old):
                    if board[y_old-i][x_old+i] != None:
                        blocker = True
            if not blocker:
                return True

    elif piece_type == 'Q': # All Queen Validation
        if x_old == x_new and y_old < y_new:
            blocker = False
            for i in range(y_old+1,y_new):
                if board[i][x_old] != None:
                    blocker = True
            if not blocker:
                return True

        elif x_old == x_new and y_old > y_new:
            blocker = False
            for i in range(y_new+1,y_old):
                if board[i][x_old] != None:
                    blocker = True
            if not blocker:
                return True

        elif y_old == y_new and x_old < x_new:
            blocker = False
            for i in range(x_old+1,x_new):
                if board[y_old][i] != None:
                    blocker = True
            if not blocker:
                return True

        elif y_old == y_new and x_old > x_new:
            blocker = False
            for i in range(x_new+1,x_old):
                if board[y_old][i] != None:
                    blocker = True
            if not blocker:
                return True

        elif x_old - x_new == y_old - y_new:
            blocker = False
```

```
            if x_old > x_new:
                for i in range(1,x_old-x_new):
                    if board[y_old-i][x_old-i] != None:
                        blocker = True
            else:
                for i in range(1,x_new-x_old):
                    if board[y_old+i][x_old+i] != None:
                        blocker = True
            if not blocker:
                return True
        elif x_old - x_new == -(y_old - y_new):
            blocker = False
            if x_old > x_new:
                for i in range(1,x_old-x_new):
                    if board[y_old+i][x_old-i] != None:
                        blocker = True
            else:
                for i in range(1,x_new-x_old):
                    if board[y_old-i][x_old+i] != None:
                        blocker = True
            if not blocker:
                return True

    elif piece_type == 'K': # All King Validation
        if abs(x_new - x_old) <= 1 and abs(y_new-y_old) <= 1:
            return True

    return False

def check_for_check(self, king, board_to_use=None, display=False): # Checks
for check for the passed king
    self.check_pos()
    if board_to_use == None:
        board_to_use = self.__board_array

    location  = king.get_location()
    x, y = self.calculate_coordinates(location)
    colour = king.get_colour()

    # Pawn for Black King
    if colour == 'B':
        try:
            if board_to_use[y+1][x-1] != None and y+1 <= 7 and x-1 >= 0:
                if board_to_use[y+1][x-1].get_type() == 'P' and board_to_use[y+1][x-
1].get_colour() == 'W':
                    if display:
                        print(f'Pawn at {board_to_use[y+1][x-1].get_location()} is
checking king')
```

```
                    #print(f'Pawn at {board_to_use[y+1][x-1].get_location()} is checking
king')
                    return True
          except: pass
          try:
             if  board_to_use[y+1][x+1] != None and y+1 <= 7 and x+1 <= 7:
                if board_to_use[y+1][x+1].get_type() == 'P' and
board_to_use[y+1][x+1].get_colour() == 'W':
                    if display:
                       print(f'Pawn at {board_to_use[y+1][x+1].get_location()} is
checking king')
                    #print(f'Pawn at {board_to_use[y+1][x+1].get_location()} is checking
king')
                    return True
          except: pass
       # Pawn for White King
       if colour == 'W':
          try:
             if board_to_use[y-1][x-1] != None and y-1 >= 0 and x-1 >= 0:
                if board_to_use[y-1][x-1].get_type() == 'P' and board_to_use[y-1][x-
1].get_colour() == 'B':
                    if display:
                       print(f'Pawn at {board_to_use[y-1][x-1].get_location()} is checking
king')
                    #print(f'Pawn at {board_to_use[y-1][x-1].get_location()} is checking
king')
                    return True
          except: pass
          try:
             if  board_to_use[y-1][x+1] != None and y-1 >= 0 and x+1 <= 7:
                if board_to_use[y-1][x+1].get_type() == 'P' and board_to_use[y-
1][x+1].get_colour() == 'B':
                    if display:
                       print(f'Pawn at {board_to_use[y-1][x+1].get_location()} is
checking king')
                    #print(f'Pawn at {board_to_use[y-1][x+1].get_location()} is checking
king')
                    return True
          except: pass
       # Rook (+queen)
       row = y
       while row < 7:
          row += 1
          square = board_to_use[row][x]
          if square != None:
             if square.get_colour() == colour:
                break
             else:
```

```
                    if square.get_type() == 'R' or square.get_type() == 'Q':
                        if display:
                            print(f'Rook (or queen) at {square.get_location()} is checking
king')
                        return True
                    else:
                        break
        row = y
        while row > 0:
            row -= 1
            square = board_to_use[row][x]
            if square != None:
                if square.get_colour() == colour:
                    break
                else:
                    if square.get_type() == 'R' or square.get_type() == 'Q':
                        if display:
                            print(f'Rook (or queen) at {square.get_location()} is checking
king')
                        return True
                    else:
                        break
        column = x
        while column < 7:
            column += 1
            square = board_to_use[y][column]
            if square != None:
                if square.get_colour() == colour:
                    break
                else:
                    if square.get_type() == 'R' or square.get_type() == 'Q':
                        if display:
                            print(f'Rook (or queen) at {square.get_location()} is checking
king')
                        return True
                    else:
                        break
        column = x
        while column > 0:
            column -= 1
            square = board_to_use[y][column]
            if square != None:
                if square.get_colour() == colour:
                    break
                else:
                    if square.get_type() == 'R' or square.get_type() == 'Q':
                        if display:
```

```
                        print(f'Rook (or queen) at {square.get_location()} is checking
king')
                    return True
                else:
                    break
        # Bishop (+queen)
        row = y
        column = x
        while row < 7 and column < 7:
            row += 1
            column += 1
            square = board_to_use[row][column]
            if square != None:
                if square.get_colour() == colour:
                    break
                else:
                    if square.get_type() == 'B' or square.get_type() == 'Q':
                        if display:
                            print(f'Bishop (or queen) at {square.get_location()} is checking
king')
                        return True
                    else:
                        break
        row = y
        column = x
        while row > 0 and column > 0:
            row -= 1
            column -= 1
            square = board_to_use[row][column]
            if square != None:
                if square.get_colour() == colour:
                    break
                else:
                    if square.get_type() == 'B' or square.get_type() == 'Q':
                        if display:
                            print(f'Bishop (or queen) at {square.get_location()} is checking
king')
                        return True
                    else:
                        break
        row = y
        column = x
        while row > 0 and column < 7:
            row -= 1
            column += 1
            square = board_to_use[row][column]
            if square != None:
                if square.get_colour() == colour:
```

```
                    break
                else:
                    if square.get_type() == 'B' or square.get_type() == 'Q':
                        if display:
                            print(f'Bishop (or queen) at {square.get_location()} is checking
king')
                        return True
                    else:
                        break
        row = y
        column = x
        while row < 7 and column > 0:
            row += 1
            column -= 1
            square = board_to_use[row][column]
            if square != None:
                if square.get_colour() == colour:
                    break
                else:
                    if square.get_type() == 'B' or square.get_type() == 'Q':
                        if display:
                            print(f'Bishop (or queen) at {square.get_location()} is checking
king')
                        return True
                    else:
                        break
        # Knight
        spaces = [[2,1],[1,2],[-1,2],[-2,1],[-2,-1],[-1,-2],[1,-2],[2,-1]]
        for space in spaces:
            yAdd = space[0]
            xAdd = space[1]
            if xAdd+x <= 7 and xAdd+x >=0 and yAdd+y <= 7 and yAdd+y >= 0:
                square = board_to_use[y+yAdd][x+xAdd]
                if square != None:
                    if square.get_type() == 'N' and square.get_colour() != colour:
                        if display:
                            print(f'Knight at {square.get_location()} is checking king')
                        #print(f'Knight at {square.get_location()} is checking king')
                        return True

        # King on King
        spaces  = [[-1,-1],[-1,0],[-1,1],[0,1],[1,1],[1,0],[1,-1],[0,-1]]
        for space in spaces:
            yAdd = space[0]
            xAdd = space[1]
            if xAdd+x <= 7 and xAdd+x >=0 and yAdd+y <= 7 and yAdd+y >= 0:
                square = board_to_use[y+space[0]][x+space[1]]
                if square != None:
```

```
                if square.get_type() == 'K' and square.get_colour() != colour:
                    if display:
                        print(f'King at {square.get_location()} is checking king')
                    return True
        return False

    def promotion_check(self,x,y):
        piece = self.__board_array[y][x]

        if (y == 7 or y == 0) and piece.get_type() == 'P':
            return True, piece
        else:
            return False, None

    def valid_moves(self, colour, board=None, check=False, order=False,
des=True,display=False): # Returns valid moves in the form
[[x_old,y_old,x_new,y_new],[x_old,y_old,x_new,y_new],...]
        valid_moves = []
        if board == None:
            board = self.__board_array

        for y in range(8):
            for x in range(8):
                square = board[y][x]
                if square != None:
                    t = square.get_type()
                    c = square.get_colour()
                    if c == colour:
                        if t == 'P':
                            if c == 'W':
                                try:
                                    if board[y-1][x] == None and y-1 >= 0:
                                        valid_moves.append([x,y,x,y-1])
                                except: pass
                                try:
                                    if board[y-1][x-1] != None and y-1>=0 and x-1>=0:
                                        if board[y-1][x-1].get_colour() != c:
                                            valid_moves.append([x,y,x-1,y-1])
                                except: pass
                                try:
                                    if board[y-1][x+1] != None and y-1>=0 and x+1<=7:
                                        if board[y-1][x+1].get_colour() != c:
                                            valid_moves.append([x,y,x+1,y-1])
                                except: pass
                                try:
                                    if board[y-2][x] == None and board[y-1][x] == None and
y==6:
                                        valid_moves.append([x,y,x,y-2])
```

```
                        except: pass
                        try:
                        # EN PASSANT
                            if board[y][x-1] != None:
                                if board[y][x-1].get_type() == 'P' and board[y][x-
1].get_colour() != c and board[y][x-1].get_en_passant():
                                    valid_moves.append([x,y,x-1,y-1])
                        except: pass
                        try:
                            if board[y][x+1] != None:
                                if board[y][x+1].get_type() == 'P' and
board[y][x+1].get_colour() != c and board[y][x+1].get_en_passant():
                                    valid_moves.append([x,y,x+1,y-1])
                        except: pass

                    if c == 'B':
                        try:
                            if board[y+1][x] == None and y+1 <= 7:
                                valid_moves.append([x,y,x,y+1])
                        except: pass
                        try:
                            if board[y+1][x-1] != None and y+1<=7 and x-1>=0:
                                if board[y+1][x-1].get_colour() != c:
                                    valid_moves.append([x,y,x-1,y+1])
                        except: pass
                        try:
                            if board[y+1][x+1] != None and y+1<=7 and x+1<=7:
                                if board[y+1][x+1].get_colour() != c:
                                    valid_moves.append([x,y,x+1,y+1])
                        except: pass
                        try:
                            if board[y+2][x] == None and board[y+1][x] == None and
y==1:
                                valid_moves.append([x,y,x,y+2])
                        except: pass
                        try:
                        # EN PASSANT
                            if board[y][x-1] != None:
                                if board[y][x-1].get_type() == 'P' and board[y][x-
1].get_colour() != c and board[y][x-1].get_en_passant():
                                    valid_moves.append([x,y,x-1,y+1])
                        except: pass
                        try:
                            if board[y][x+1] != None:
                                if board[y][x+1].get_type() == 'P' and
board[y][x+1].get_colour() != c and board[y][x+1].get_en_passant():
                                    valid_moves.append([x,y,x+1,y+1])
                        except: pass
```

```
elif t == 'R' or t == 'Q':
    for i in range(x+1,8):
        if board[y][i] == None:
            valid_moves.append([x,y,i,y])
        elif board[y][i].get_colour() != c:
            valid_moves.append([x,y,i,y])
            break
        else:
            break

    for i in range(x-1,-1,-1):
        if board[y][i] == None:
            valid_moves.append([x,y,i,y])
        elif board[y][i].get_colour() != c:
            valid_moves.append([x,y,i,y])
            break
        else:
            break

    for i in range(y+1,8):
        if board[i][x] == None:
            valid_moves.append([x,y,x,i])
        elif board[i][x].get_colour() != c:
            valid_moves.append([x,y,x,i])
            break
        else:
            break

    for i in range(y-1,-1,-1):
        if board[i][x] == None:
            valid_moves.append([x,y,x,i])
        elif board[i][x].get_colour() != c:
            valid_moves.append([x,y,x,i])
            break
        else:
            break

if t == 'B' or t == 'Q':
    row = y
    column = x
    while row < 7 and column < 7:
        row += 1
        column += 1
        if board[row][column] == None:
            valid_moves.append([x,y,column,row])
        elif board[row][column].get_colour() != c:
            valid_moves.append([x,y,column,row])
```

```
                break
            else:
                break

        row = y
        column = x
        while row > 0 and column < 7:
            row -= 1
            column += 1
            if board[row][column] == None:
                valid_moves.append([x,y,column,row])
            elif board[row][column].get_colour() != c:
                valid_moves.append([x,y,column,row])
                break
            else:
                break

        row = y
        column = x
        while row > 0 and column > 0:
            row -= 1
            column -= 1
            if board[row][column] == None:
                valid_moves.append([x,y,column,row])
            elif board[row][column].get_colour() != c:
                valid_moves.append([x,y,column,row])
                break
            else:
                break

        row = y
        column = x
        while row < 7 and column > 0:
            row += 1
            column -= 1
            if board[row][column] == None:
                valid_moves.append([x,y,column,row])
            elif board[row][column].get_colour() != c:
                valid_moves.append([x,y,column,row])
                break
            else:
                break

    elif t == 'N':
        spaces = [[2,1],[1,2],[-1,2],[-2,1],[-2,-1],[-1,-2],[1,-2],[2,-1]]
        for space in spaces:
            y_add = space[0]
            x_add = space[1]
```

```
                    if x_add+x <= 7 and x_add+x >=0 and y_add+y <= 7 and
y_add+y >= 0:
                        if board[y_add+y][x_add+x] == None:
                            valid_moves.append([x,y,x_add+x,y_add+y])
                        elif board[y_add+y][x_add+x].get_colour() != c:
                            valid_moves.append([x,y,x_add+x,y_add+y])

            elif t == 'K':
                spaces  = [[-1,-1],[-1,0],[-1,1],[0,1],[1,1],[1,0],[1,-1],[0,-1]]
                for space in spaces:
                    y_add = space[0]
                    x_add = space[1]
                    if x_add+x <= 7 and x_add+x >=0 and y_add+y <= 7 and
y_add+y >= 0:
                        if board[y_add+y][x_add+x] == None:
                            valid_moves.append([x,y,x_add+x,y_add+y])
                        elif board[y_add+y][x_add+x].get_colour() != c:
                            valid_moves.append([x,y,x_add+x,y_add+y])

            if t == 'R' and board[y][4] != None:
                valid = False
                if x==0 and board[y][4].get_type() == 'K':
                    valid = True
                    for check_x in range(1,4):
                        if board[y][check_x] != None:
                            valid = False
                    king = board[y][4]
                    if self.check_for_check(king,board):
                        valid = False
                    if king.get_moved():
                        valid = False
                elif x==7 and board[y][4].get_type() == 'K':
                    valid = True
                    for check_x in range(5,7):
                        if board[y][check_x] != None:
                            valid = False
                    king = board[y][4]
                    if self.check_for_check(king,board):
                        valid = False
                    if king.get_moved():
                        valid = False
                if square.get_moved():
                    valid = False
                if valid:
                    valid_moves.append([x,y,None,None])

        for move in valid_moves:
            try:
```

```
                if move[0] < 0 or move[0] > 7 or move[1] < 0 or move[1] > 7 or move[2] <
0 or move[2] > 7 or move[3] < 0 or move[3] > 7:
                    valid_moves.remove(move)
            except: pass

        if check:
            valid_moves_old = copy.deepcopy(valid_moves)
            for move in valid_moves_old:
                new_board = copy.deepcopy(board)
                if move[2] != None:
                    new_board[move[3]][move[2]] = new_board[move[1]][move[0]]
                    new_board[move[1]][move[0]] = None

new_board[move[3]][move[2]].change_location(self.calculate_coordinates(x=move
[2],y=move[3]))
                else:
                    if move[0] == 0:
                        new_board[move[1]][3] = new_board[move[1]][move[0]]
                        new_board[move[1]][3].change_location(f'c{abs(move[1]-8)}')
                        new_board[move[1]][move[0]] = None
                        new_board[move[1]][2] = new_board[move[1]][4]
                        new_board[move[1]][2].change_location(f'd{abs(move[1]-8)}')
                        new_board[move[1]][4] = None
                    else:
                        new_board[move[1]][5] = new_board[move[1]][move[0]]
                        new_board[move[1]][5].change_location(f'f{abs(move[1]-8)}')
                        new_board[move[1]][move[0]] = None
                        new_board[move[1]][6] = new_board[move[1]][4]
                        new_board[move[1]][6].change_location(f'g{abs(move[1]-8)}')
                        new_board[move[1]][4] = None

                new_king = False
                for row in new_board:
                    for pieceCol in row:
                        if pieceCol != None:
                            if pieceCol.get_type() == 'K' and pieceCol.get_colour() == colour:
                                new_king = pieceCol
                                break
                    if new_king != False:
                        break

                if new_king == False:
                    print(f'''FATAL ERROR: NEW KING NOT FOUND
                          BOARD:''')
                    self.display_board(board_to_use=new_board)

                if self.check_for_check(new_king,new_board):
                    if display:
```

```
                    print(f"Move {move} is invalid as it puts king in check")
                valid_moves.remove(move)

        if order:
            score_array = []
            new_valid_moves = []
            for move in valid_moves:
                new_board = self.sim_move(move,board)
                score =
self.evaluate(colour=colour,board=new_board,check=False,checkmate=False)
                done = False
                for score_i in score_array:
                    if done:
                        break
                    index = score_array.index(score_i)
                    if des:
                        if score_i < score:
                            score_array.insert(index,score)
                            new_valid_moves.insert(index,move)
                            done = True
                    else:
                        if score_i > score:
                            score_array.insert(index,score)
                            new_valid_moves.insert(index,move)
                            done = True
                if not done:
                    score_array.append(score)
                    new_valid_moves.append(move)
            valid_moves = new_valid_moves

        return valid_moves

    def get_board(self):
        return self.__board_array

    def check_pos(self):
        for y in range(8):
            for x in range(8):
                square = self.__board_array[y][x]
                if square != None:
                    x1,y1 = self.calculate_coordinates(square.get_location())
                    if x != x1 or y != y1:
                        print(f'{square.get_colour()}{square.get_type()} at {x},{y} thinks its at
{square.get_location()}')
                        square.change_location(self.calculate_coordinates(x=x,y=y))

    def evaluate(self,colour='W',board=None,check=False,checkmate=False): #
Colour passed represents positive score
```

```
        POSSETION_WIEGHT = 0
        POSSETION_BONUS = 0.1
        PAWN_ADVANCEMENT_WIEGHT = 0.2
        CHECK_WIEGHT = 0
        CHECKMATE_WIEGHT = 9999
        STALEMATE_WIEGHT = -1
        PAWN_CENTRE_WIEGHT = 1

        score = 0
        if board == None:
            board = self.__board_array

        for row in board:
            for square in row:
                if square != None:
                    if square.get_type() == 'P':
                        if square.get_colour() == colour:
                            score += 1 + POSSETION_WIEGHT+POSSETION_BONUS
                            if board.index(row) >= 3 and board.index(row) <= 4 and
row.index(square) >= 3 and row.index(square) <= 4:
                                score += PAWN_CENTRE_WIEGHT
                            if colour == 'W':
                                score += (6-board.index(row)) *
PAWN_ADVANCEMENT_WIEGHT
                            else:
                                score += (board.index(row)-1) *
PAWN_ADVANCEMENT_WIEGHT
                        else:
                            score -= 1 + POSSETION_WIEGHT
                            if board.index(row) >= 3 and board.index(row) <= 4 and
row.index(square) >= 3 and row.index(square) <= 4:
                                score -= PAWN_CENTRE_WIEGHT
                            if square.get_colour() == 'W':
                                score -= (6-board.index(row)) *
PAWN_ADVANCEMENT_WIEGHT
                            else:
                                score -= (board.index(row)-1) *
PAWN_ADVANCEMENT_WIEGHT
                    elif square.get_type() == 'R':
                        if square.get_colour() == colour:
                            score += 5 + POSSETION_WIEGHT+POSSETION_BONUS
                        else:
                            score -= 5 + POSSETION_WIEGHT
                    elif square.get_type() == 'N':
                        if square.get_colour() == colour:
                            score += 3 + POSSETION_WIEGHT+POSSETION_BONUS
                        else:
                            score -= 3 + POSSETION_WIEGHT
```

```
                elif square.get_type() == 'B':
                    if square.get_colour() == colour:
                        score += 3 + POSSETION_WIEGHT+POSSETION_BONUS
                    else:
                        score -= 3 + POSSETION_WIEGHT
                elif square.get_type() == 'Q':
                    if square.get_colour() == colour:
                        score += 10 + POSSETION_WIEGHT+POSSETION_BONUS
                    else:
                        score -= 10 + POSSETION_WIEGHT
                elif square.get_type() == 'K':
                    if square.get_colour() == colour:
                        score += 100 + POSSETION_WIEGHT+POSSETION_BONUS
                    else:
                        score -= 100 + POSSETION_WIEGHT

                    if check:
                        if self.check_for_check(square,board):
                            if square.get_colour() == colour:
                                score -= CHECK_WIEGHT
                                if checkmate:
                                    if
self.valid_moves(square.get_colour(),board,check=True)==[]:
                                        score -= CHECKMATE_WIEGHT
                            else:
                                score += CHECK_WIEGHT
                                if checkmate:
                                    if
self.valid_moves(square.get_colour(),board,check=True)==[]:
                                        score += CHECKMATE_WIEGHT
                        else:
                            if square.get_colour() == colour:
                                if
self.valid_moves(square.get_colour(),board,check=True)==[]:
                                    score += STALEMATE_WIEGHT
                            else:
                                if
self.valid_moves(square.get_colour(),board,check=True)==[]:
                                    score -= STALEMATE_WIEGHT

        return score

    def sim_move(self,move,board):
        new_board = copy.deepcopy(board)
        if move[2] != None:
            new_board[move[3]][move[2]] = new_board[move[1]][move[0]]
            new_board[move[1]][move[0]] = None
```

```
new_board[move[3]][move[2]].change_location(self.calculate_coordinates(x=move
[2],y=move[3]))

            if (move[3] == 0 or move[3] == 7) and
new_board[move[3]][move[2]].get_type() == 'P':
                new_board[move[3]][move[2]].promote('Q')
        else:
            if move[0] == 0:
                new_board[move[1]][3] = new_board[move[1]][move[0]]
                new_board[move[1]][3].change_location(f'd{abs(move[1]-8)}')
                new_board[move[1]][move[0]] = None
                new_board[move[1]][2] = new_board[move[1]][4]
                new_board[move[1]][2].change_location(f'c{abs(move[1]-8)}')
                new_board[move[1]][4] = None
            else:
                new_board[move[1]][5] = new_board[move[1]][move[0]]
                new_board[move[1]][5].change_location(f'f{abs(move[1]-8)}')
                new_board[move[1]][move[0]] = None
                new_board[move[1]][6] = new_board[move[1]][4]
                new_board[move[1]][6].change_location(f'g{abs(move[1]-8)}')
                new_board[move[1]][4] = None
        return new_board

class Piece():
    def __init__(self, piece_type, location, colour):
        self.__type = piece_type
        self.__location = location
        self.__colour = colour
        self.__en_passant_sus = False # True when pawn is suseptable to en
passant
        self.__moved = False

    def get_colour(self):
        return self.__colour

    def get_type(self):
        return self.__type

    def get_location(self):
        return self.__location

    def change_location(self, coords):
        self.__location = coords

    def promote(self, piece):
        if self.__type == 'P':
            self.__type = piece
```

```
    def register_moved(self):
        self.__moved = True

    def amend_en_passant(self, status):
        if status != self.__en_passant_sus:
            self.__en_passant_sus = status

    def get_en_passant(self):
        return self.__en_passant_sus

    def get_moved(self):
        return self.__moved

class Player(abc.ABC):
    def __init__(self, colour):
        self.__colour = colour

    def get_colour(self):
        return self.__colour

    def get_move(self):
        return

    def get_type(self):
        return None

class User(Player):
    def __init__(self, colour):
        super().__init__(colour)

    def get_type(self):
        return 'User'

class Computer(Player):
    def __init__(self, colour, difficulty, depth):
        super().__init__(colour)
        self.__difficulty = difficulty
        self.__colour = colour
        self.__DEPTH = depth

        self.__initial_depth = -1

    def get_move(self, board_to_use,board):
        if self.__difficulty == 0:
            print('Random move')
            return self.__random_move(board_to_use,board)
        elif self.__difficulty == 1:
```

```
            print('Basic move')
            return self.__basic_move(board_to_use,board)
        elif self.__difficulty == 2:
            print('Advanced move')
            return self.__adv_move(board_to_use,board)

    def __random_move(self, board_to_use,board):
        valid_moves =
board.valid_moves(self.__colour,board=board_to_use,check=True)
        chosen = random.choice(valid_moves)
        return chosen[0],chosen[1],chosen[2],chosen[3]

    def __basic_move(self,board_to_use,board):
        valid_moves =
board.valid_moves(self.__colour,board=board_to_use,check=True)
        top_score = float('-inf')
        top_move = None
        top_moves = []
        for move in valid_moves:
            new_board = board.sim_move(move,board_to_use)
            score =
board.evaluate(self.__colour,new_board,check=True,checkmate=True)
            if score > top_score:
                top_score = score
                top_move = move
                top_moves = [top_move]
            elif score == top_score:
                top_moves.append(move)
        if len(top_moves) > 1:
            top_move = random.choice(top_moves)
        return top_move[0],top_move[1],top_move[2],top_move[3]

    def __adv_move(self,board_to_use,board):
        if self.__DEPTH == -1:
            length = len(board.valid_moves(self.__colour,check=True))
            if length <= 10:
                depth = 3
            else:
                depth = 2
        else:
            depth = self.__DEPTH
        self.__initial_depth = depth
        timeStart = time.time()
        score,top_move = self.__min_max(board_to_use,board,depth,float('-
inf'),float('+inf'))
        print()
        print(f'DEPTH: {depth}')
        print(f'TOP SCORE: {score}')
```

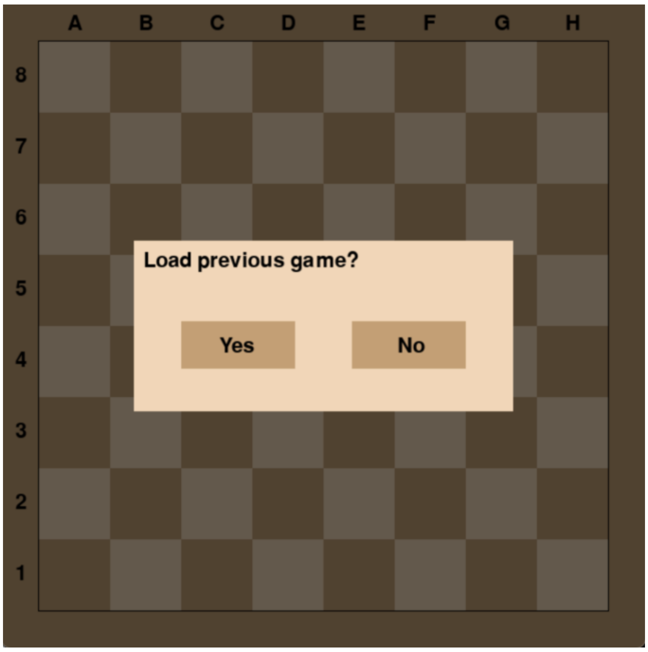
```
        print(f'TIME TAKEN: {time.time()-timeStart}')
        return top_move[0],top_move[1],top_move[2],top_move[3]

    def __min_max(self,board_to_use,board,depth,alpha,beta,maximising=True): #
maximising is true for maximising and false for minimising. SHOULD ALWAYS
START AS TRUE
        if depth == 0:
            return
board.evaluate(colour=self.__colour,board=board_to_use,check=True,checkmate
=True), None

        if maximising:
            max_score = float('-inf')
            max_score_move = None

            order = False
            if depth > 1:
                order = True

            possible_moves = board.valid_moves(self.__colour, board=board_to_use,
check=True,order=order,des=True)
            for move in possible_moves:
                if self.__initial_depth == depth:
                    index = possible_moves.index(move)
                    length = len(possible_moves)
                    print('|'+('#'*(index+1))+('-'*(length-(index+1)))+'|',end='\r')

                new_board = board.sim_move(move,board_to_use)
                score,m = self.__min_max(new_board,board,depth-
1,alpha,beta,maximising=False)

                if score > max_score:
                    max_score = score
                    max_score_move = move
                if alpha<max_score:
                    alpha = max_score
                if beta <= alpha:
                    break
            if max_score_move == None:
                return
board.evaluate(colour=self.__colour,board=board_to_use,check=True,checkmate
=True), None
            return max_score,max_score_move
        else:
            min_score = float('+inf')
            min_score_move = None

            if self.__colour == 'W':
```

```
            colour = 'B'
        else:
            colour = 'W'

        order = False
        if depth > 1:
            order = True

        possible_moves = board.valid_moves(colour, board=board_to_use,
check=True,order=order,des=False)
        for move in possible_moves:
            new_board = board.sim_move(move,board_to_use)

            score,m = self.__min_max(new_board,board,depth-
1,alpha,beta,maximising=True)
            if score < min_score:
                min_score = score
                min_score_move = move
            if beta>min_score:
                beta = min_score
            if beta <= alpha:
                break
        if min_score_move == None:
            return
board.evaluate(colour=self.__colour,board=board_to_use,check=True,checkmate
=True), None
        return min_score,min_score_move

    def get_type(self):
        return 'Computer'

def MainProgram():
    main_board = Game()

if __name__ == '__main__':
    MainProgram()
```

Screen Grabs

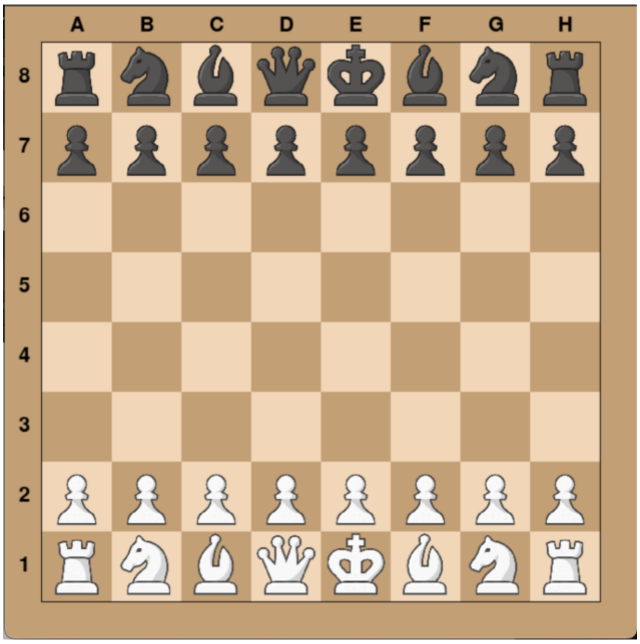
These are screen grabs of the final UI

Load Game Menu:



Choose Difficulty Menu:

Board Screen:



Promotion Screen:

Checkmate Screen:

Stalemate Screen: