AI-Powered Offline Chess

By Evelyn Jane Sutjiadi (2802501054) L1AC

Introduction

Lately, I have been playing chess often, whether it is on my way to campus, during breaks, or in class. With many ideas in mind for my final algorithm and programming project, I made a lot of demo projects, but for me, it was just not impressive enough, or just too simple to do. After a random 3 AM thought on chess tactics, I just came up with a solution to make an AI-powered chess on my own, where I can play chess with a robot with no internet connection.

Project Goal

The goal of this project was to design and implement an AI capable of playing chess at a competitive level. The AI put up a lot of techniques such as board evaluation, move generation, and decision-making using the minimax algorithm with alpha-beta pruning, which was used in my project. The project showcases a systematic approach to game AI, offering a well-balanced strategy gameplay with high move efficiency.

Project Specification

Main Objectives & Features

The main objectives & features of the project are :

1. Develop a playable chess game interface
   A Python-based graphical interface allows users to play chess against the AI, with the user always being the white-colored pieces while the AI always being the black-colored pieces with a turn-based gameplay with white being first and black next like standard chess rules.
2. Implement chess rules and mechanics
   These rules and mechanics include legal move generation, special moves (for example, castling & en passant), and also game state validation.
3. AI Opponent Design
   This AI would do things that a normal human opponent would do like for example evaluating board positions, predicting future outcomes and possibilities, and also selecting moves that would maximize its chances of winning the game.

Instructions :

1. Download all the files, or download the folder then extract the files
2. Make sure to have all the codes in a folder called 'Chess'
3. In the folder there should be 3 python files and 1 folder called 'chess/images'
4. The folder should be Chess/ chessmain.py, ChessEngine.py, ChessAI.py, Chess/Images (the folder)

5. Run the the chessmain.py file, and the program should be working

How to play : Simply click the pieces you want to move, there should be highlighted spots on where you can move that specific piece. Press 'Z' in your keyboard to undo a move.

Solution Design

Architecture Overview

The project was divided into three main components / files :

1. chessmain.py
2. ChessEngine.py
3. ChessAI.py

Chessmain.py

This file handles the overall graphical interface and interactions between the user and the game state.

```
2    import pygame as p
3    import ChessEngine
4    import ChessAI
5    import random
```

Figure 1 : Imports & Initialization

```
7    width = height = 512 #512x512
8    dimension = 8 #8x8 dimensions
9    sq_size = height // dimension #square size = height / dimension
10   max_fps = 15 #for animations
11   images = {}
```

This code from lines 2 - 5 imports and initializes the basic setup for a chess game using Pygame. First, the necessary modules are imported: pygame for graphical rendering and event handling, *ChessEngine* for managing the game's logic, *ChessAI* for AI functionalities, and random for randomizations where needed. The variables width and height are both set to 512, defining the size of the game window as a square of 512x512 pixels. The dimension variable is set to 8, reflecting the 8x8 chessboard grid. The *sq_size* variable calculates the size of each square on the board by dividing the height by the dimension, ensuring uniform square sizes. The variable *max_fps* is set to 15, defining the maximum frames per second to control animation speed and ensure smooth gameplay. Finally, images is initialized as an empty dictionary, which will later store the chess piece images for rendering on the board.

Main Function

```
14  def main():
15      p.init()
16      screen = p.display.set_mode((width, height))
17      clock = p.time.Clock()
18      screen.fill(p.Color("white"))
19      gs = ChessEngine.GameState()
20      validmoves = gs.getvalidmoves()
21      movemade = False #flag variable when a move is made
22      animate = False  # flag for when we should animate a move
23      loadimages() # only do this once before while loop
24      gameOver = False
25      playerOne = True  # True if human is playing white
26      playerTwo = False  # False if AI is playing black
```

Figure 3 : Full Main Function

```
14  def main():
15      p.init()
16      screen = p.display.set_mode((width, height))
17      clock = p.time.Clock()
18      screen.fill(p.Color("white"))
```

Figure 4 : Main Function

This function, *main()*, sets up the initial environment for the chess game using Pygame. The *p.init()* call initializes all the Pygame modules to prepare for game execution. The screen variable is created using *p.display.set_mode((width, height))* seen in line 16, which sets up the display window with dimensions defined earlier (512x512 pixels). The clock variable is instantiated using *p.time.Clock()*, enabling control over the game's frame rate for smooth rendering and animations. Finally, the *screen.fill(p.Color("white")*) line fills the entire game window with a white background color, providing a clean slate upon which the chessboard and pieces will later be drawn for the game interface.

```
19      gs = ChessEngine.GameState()
20      validmoves = gs.getvalidmoves()
21      movemade = False #flag variable when a move is made
22      animate = False  # flag for when we should animate a move
```

Figure 5 : Game State Initialization

This snippet initializes variables needed for managing the chess game's state and user interactions. *gs = ChessEngine.GameState()* in line 19 creates an instance of the GameState class from the ChessEngine module, representing the current state of the chessboard and game logic. *validmoves = gs.getvalidmoves()* retrieves all possible moves for the current player based on the game state. *movemade = False* initializes a flag variable to track whether a move has been made; this helps determine if updates to the game state or graphics are needed. Line 22, animate = False initializes another flag used to decide whether to animate the most recent move, enhancing the visual experience when a move occurs. Together, these variables facilitate game logic and rendering updates.

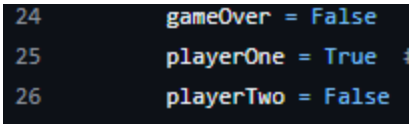


Figure 6 : Player Settings

Lines 24 - 26, player settings, initializes game state variables to manage gameplay dynamics. *gameOver = False* indicates the game is active and hasn't ended due to checkmate, stalemate, or other conditions. *playerOne = True* signifies that the white pieces are controlled by a human, while *playerTwo = False* specifies that the black pieces are controlled by an AI. These flags help determine player turns and control logic during the game.

Main Game Loop

```
28          running = True
29          sqselected = () #no square is selected initially, keep track with the last click of the user (tuple : row, column)
30          playerclicks = [] #keep track of the player clicks (2 tuples : r6, c4)
31
32          while running:
33              humanTurn = (gs.whiteToMove and playerOne) or (not gs.whiteToMove and playerTwo)
34
35              for e in p.event.get():
36                  if e.type == p.QUIT:
37                      running = False
38
39                  # Mouse handler
40                  elif e.type == p.MOUSEBUTTONDOWN:
41                      if not gameOver:
42                          if humanTurn:
43                              location = p.mouse.get_pos() #(x,y) location of the mouse
44                              col = location[0]//sq_size
45                              row = location[1]//sq_size
46
47                              if sqselected == (row,col): #user clicked the same square twice
48                                  sqselected = () #deselect
49                                  playerclicks = [] #clear clicks
50                              else:
51                                  sqselected = (row,col)
52                                  playerclicks.append(sqselected) #append for both 1st and 2nd clicks
53                              if len(playerclicks) == 2: #after 2nd click
54                                  move = ChessEngine.move(playerclicks[0], playerclicks[1], gs.board)
55                                  for i in range(len(validmoves)):
56                                      if move == validmoves[i]:
57                                          gs.makemove(validmoves[i])
58                                          movemade = True
59                                          animate = True
60                                          sqselected = () #reset user clicks
61                                          playerclicks = []
62                                  if not movemade:
63                                      playerclicks = [sqselected]
64                      else:  # Game is over, any click resets
65                          gs = ChessEngine.GameState()
66                          validmoves = gs.getvalidmoves()
67                          sqselected = ()
68                          playerclicks = []
69                          movemade = False
70                          animate = False
71                          gameOver = False
```

```
            #key handlers
            elif e.type == p.KEYDOWN:
                if e.key == p.K_z: #undo when z is pressed
                    gs.undomove()
                    movemade = True
                    animate = False
                    gameOver = False
                elif e.key == p.K_r:  # reset the game when 'r' is pressed
                    gs = ChessEngine.GameState()
                    validmoves = gs.getvalidmoves()
                    sqselected = ()
                    playerclicks = []
                    movemade = False
                    animate = False
                    gameOver = False

        # AI move finder
        if not gameOver and not humanTurn:
            ai_move = ChessAI.find_best_move(gs)
            if ai_move is None:
                ai_move = random.choice(validmoves)
            gs.makemove(ai_move)
            movemade = True
            animate = True

        if movemade:
            if animate:
                animatemove(gs.moveLog[-1], screen, gs.board, clock)
            validmoves = gs.getvalidmoves()
            movemade = False
            animate = False

        drawGameState(screen, gs, sqselected, validmoves)

        # Check for checkmate or stalemate
        if not gameOver and len(validmoves) == 0:
            gameOver = True
            drawGameState(screen, gs, sqselected, validmoves)  # Redraw final position
            if gs.is_in_check():
                if gs.whiteToMove:
                    drawEndGameText(screen, "Black Wins!")
                else:
                    drawEndGameText(screen, "White Wins!")
            else:
                drawEndGameText(screen, "Stalemate!")
            p.display.flip()  # Update the display with the final position and message
            # Wait for player input to reset
            waiting = True
```

```
121                 while waiting and running:
122                     for event in p.event.get():
123                         if event.type == p.QUIT:
124                             running = False
125                             waiting = False
126                         elif event.type == p.MOUSEBUTTONDOWN or event.type == p.KEYDOWN:
127                             waiting = False
128                             gs = ChessEngine.GameState()
129                             validmoves = gs.getvalidmoves()
130                             sqselected = ()
131                             playerclicks = []
132                             movemade = False
133                             animate = False
134                             gameOver = False
135                 continue  # Skip the rest of the game loop after reset
136 
137             clock.tick(max_fps)
138             p.display.flip()
```

Table 1 : Full Main Game Loop

```
28          running = True
29          sqselected = () #no square is selected initially, keep track with the last click of the user (tuple : row, column)
30          playerclicks = [] #keep track of the player clicks (2 tuples : r6, c4)
31 
32          while running:
33              humanTurn = (gs.whiteToMove and playerOne) or (not gs.whiteToMove and playerTwo)
```

Figure 7 : Main Game Loop

The main game loop iinitializes variables to track the game's progress and user interactions. *running = True* ensures the main game loop continues until the player exits. *sqselected = ()* represents the initially unselected state of the board, storing the row and column of the last square clicked by the user as a tuple. *playerclicks = []* is a list that records the player's move sequence, typically storing two tuples representing the start and end positions of a piece during a move. These variables support user input and gameplay functionality.

```
35             for e in p.event.get():
36                 if e.type == p.QUIT:
37                     running = False
```

Figure 8 : Event Handling

This code processes events captured by Pygame during the game's main loop. The line for e in *p.event.get():* in line 35 iterates over all events currently in Pygame's event queue. Within the loop, the condition if *e.type == p.QUIT*: checks if the event is of type QUIT, which corresponds to the user attempting to close the game window. If such an event is detected, *running = False* sets the running variable to False, signaling that the game loop should terminate, ultimately ending the program. This ensures the game window closes properly when the user exits.

```
39                # Mouse handler
40                elif e.type == p.MOUSEBUTTONDOWN:
41                    if not gameOver:
42                        if humanTurn:
43                            location = p.mouse.get_pos() #(x,y) location of the mouse
44                            col = location[0]//sq_size
45                            row = location[1]//sq_size
```

Figure 9 : Mouse Input

This code handles mouse click events during the game. The condition elif *e.type == p.MOUSEBUTTONDOWN:* in line 40 checks if the user has clicked a mouse button. If this occurs and not *gameOver:* ensures the game is still active. Additionally, if *humanTurn:* ensures the click is processed only if it's the human player's turn. The line *location = p.mouse.get_pos()* in lines 43 retrieves the (x, y) coordinates of the mouse pointer at the time of the click. Then, *col = location[0] // sq_size* calculates the column of the chessboard square that was clicked by dividing the x-coordinate by the size of a square. Not only that, *row = location[1] // sq_size* in lines 45 computes the corresponding row using the y-coordinate. This effectively translates the screen coordinates of the mouse click to the corresponding chessboard square.

```
47                        if sqselected == (row,col): #user clicked the same square twice
48                            sqselected = () #deselect
49                            playerclicks = [] #clear clicks
50                        else:
51                            sqselected = (row,col)
52                            playerclicks.append(sqselected) #append for both 1st and 2nd clicks
```

Figure 10 : Handling Player Clicks

This code processes player clicks on the chessboard to determine selection behavior. The line if *sqselected == (row, col):* checks if the square currently selected by the user (sqselected) is the same as the one they just clicked. If so, the user has clicked the same square twice, prompting the deselection of the square by setting *sqselected = ()* to an empty tuple. Similarly, the line *playerclicks = []* clears the list that tracks the user's clicks to reset the move sequence. If the user clicks a different square, the *else: block* updates *sqselected = (row, col)* to reflect the newly selected square, preparing it for potential move input. This logic ensures user clicks are handled properly for selecting and deselecting squares.

```
53                          if len(playerclicks) == 2: #after 2nd click
54                              move = ChessEngine.move(playerclicks[0], playerclicks[1], gs.board)
55                              for i in range(len(validmoves)):
56                                  if move == validmoves[i]:
57                                      gs.makemove(validmoves[i])
58                                      movemade = True
59                                      animate = True
60                                      sqselected = () #reset user clicks
61                                      playerclicks = []
```

Figure 11 : Processing Moves

This section handles the logic after the user has made two clicks on the chessboard to select a move. The line if *len(playerclicks) == 2:* checks if the user has selected two squares, indicating a move. Once two squares are selected, a move object is created using the first and second click positions from playerclicks. The code then iterates through the list of valid moves (validmoves) to check if the move the user made matches any of the valid moves. If a match is found, *gs.makemove(validmoves[i])* is called to execute the move on the game state, and *movemade = True* flags that a move has been made. The *animate = True* flag is set to indicate that a move should be animated. After this, *sqselected = ()* resets the square selection, and *playerclicks = []* clears the list of user clicks, readying the game for the next action. This ensures that the move is processed correctly, the board state is updated, and the interface is prepared for the next user input.

```
64                      else:  # Game is over, any click resets
65                          gs = ChessEngine.GameState()
66                          validmoves = gs.getvalidmoves()
67                          sqselected = ()
68                          playerclicks = []
69                          movemade = False
70                          animate = False
71                          gameOver = False
```

Figure 12 : Game Reset

This code from line 64 - 71 only works to reset the game state when it’s over.

```
73                #key handlers
74                elif e.type == p.KEYDOWN:
75                    if e.key == p.K_z: #undo when z is pressed
76                        gs.undomove()
77                        movemade = True
78                        animate = False
79                        gameOver = False
80                    elif e.key == p.K_r:  # reset the game when 'r' is pressed
81                        gs = ChessEngine.GameState()
82                        validmoves = gs.getvalidmoves()
83                        sqselected = ()
84                        playerclicks = []
85                        movemade = False
86                        animate = False
87                        gameOver = False
```

Figure 13 : Key Input

The code from lines 73 - 87 handles keyboard input events. The *elif e.type == p.KEYDOWN:* checks if a key has been pressed. When the *K_z* key is pressed, which corresponds to the 'Z' key, the code calls *gs.undomove()* to undo the last move. This also sets *movemade = True* to indicate that a move has been undone, *animate = False* to disable animation (since there's no move to animate), and *gameOver = False* to ensure the game is still active. For the *K_r key* (which corresponds to the 'R' key), the game is reset using *gs = ChessEngine.GameState()*, which creates a new game state. The valid moves are recalculated with *validmoves = gs.getvalidmoves()*. The variables *sqselected = ()* and *playerclicks = []* are reset to clear the board state, and *movemade = False*, *animate = False*, and *gameOver = False* are also reset to prepare for a fresh game. This functionality allows the player to undo the last move or restart the game entirely, providing greater control over the game flow.

```
89                # AI move finder
90                if not gameOver and not humanTurn:
91                    ai_move = ChessAI.find_best_move(gs)
92                    if ai_move is None:
93                        ai_move = random.choice(validmoves)
94                    gs.makemove(ai_move)
95                    movemade = True
96                    animate = True
```

Figure 14 : AI Move

This code from lines 89 - 96 finds the best move using the *ChessAI.py* file or selects a random move if no best move is found. It also works as executing the AI's move.

```
98              if movemade:
99                  if animate:
100                     animatemove(gs.moveLog[-1], screen, gs.board, clock)
101                 validmoves = gs.getvalidmoves()
102                 movemade = False
103                 animate = False
104
105             drawGameState(screen, gs, sqselected, validmoves)
```

Figure 15 : Updating and Drawing

This code from lines 98 - 105 executes animations and updates the valid moves that can be done by the pieces. It also draws the updated game state (updating the board).

```
107             # Check for checkmate or stalemate
108             if not gameOver and len(validmoves) == 0:
109                 gameOver = True
110                 drawGameState(screen, gs, sqselected, validmoves)  # Redraw final position
111                 if gs.is_in_check():
112                     if gs.whiteToMove:
113                         drawEndGameText(screen, "Black Wins!")
114                     else:
115                         drawEndGameText(screen, "White Wins!")
116                 else:
117                     drawEndGameText(screen, "Stalemate!")
118                 p.display.flip()  # Update the display with the final position and message
```

Figure 16 : Game Over Check

This code form lines 107 - 118 would check if the game is over and whether there are any valid moves left for the current player, using the condition if not *gameOver* and *len(validmoves) == 0:*. If no valid moves are available and the game is not already over, it sets *gameOver = True* to mark the end of the game. Then, *drawGameState(screen, gs, sqselected, validmoves)* redraws the board, showing the final position of the pieces. The code then checks if the current player is in check using *gs.is_in_check()*. If the player is in check, it determines the winner (if it's white's turn, the message "Black Wins!") is displayed, and if it's black's turn, "White Wins!" is shown. If the game is not in check, indicating a stalemate, *drawEndGameText(screen, "Stalemate!")* is called to display the stalemate message. Finally, *p.display.flip()* updates the display to show the final position and the endgame message to the player.

```
119                 # Wait for player input to reset
120                 waiting = True
121                 while waiting and running:
122                     for event in p.event.get():
123                         if event.type == p.QUIT:
124                             running = False
125                             waiting = False
126                         elif event.type == p.MOUSEBUTTONDOWN or event.type == p.KEYDOWN:
127                             waiting = False
128                             gs = ChessEngine.GameState()
129                             validmoves = gs.getvalidmoves()
130                             sqselected = ()
131                             playerclicks = []
132                             movemade = False
133                             animate = False
134                             gameOver = False
135                 continue  # Skip the rest of the game loop after reset
136 
137             clock.tick(max_fps)
138             p.display.flip()
```

Figure 17 : Waiting for Reset After Game Over

This code from lines 119 - 138 shows that after the game ends, the program waits for the user to either close the window or click a button to reset, where they can press the screen or simply press ‘r’ on their keyboard. It would reset the game state by reinitializing the game variables.

```
140 def drawEndGameText(screen, text):
141     font = p.font.SysFont("Arial", 32, True, False)
142     textObject = font.render(text, False, p.Color("Black"))
143     textLocation = p.Rect(0, 0, width, height).move(width/2 - textObject.get_width()/2, height/2 - textObject.get_height()/2)
144     screen.blit(textObject, textLocation)
145 
146     # Add "Click to reset" message
147     resetFont = p.font.SysFont("Arial", 24, True, False)
148     resetText = resetFont.render("Click to reset", False, p.Color("Black"))
149     resetLocation = textLocation.move(0, textObject.get_height() + 10)
150     screen.blit(resetText, resetLocation)
```

Figure 18 : Helper Functions (drawGameEndText)

This code from lines 140 - 150 is the helper function for *drawGameEndText*. This function would display a message (example : "*Black Wins!*"), fetching it from line 113, at the center of the screen when the game ends. Not only that, it would add a small "*Click to reset*" message, shown in line 148 below the main text.

```
152  def animatemove(move, screen, board, clock):
153      global colors
154      coords = []  # list of coords that the animation will move through
155      dR = move.endrow - move.startrow
156      dC = move.endcol - move.startcol
157      framesPerSquare = 10  # frames to move one square
158      frameCount = (abs(dR) + abs(dC)) * framesPerSquare
159      for frame in range(frameCount + 1):
160          r = move.startrow + dR*frame/frameCount
161          c = move.startcol + dC*frame/frameCount
162          drawboard(screen)
163          drawpieces(screen, board)
164          # erase the piece moved from its ending square
165          color = colors[(move.endrow + move.endcol) % 2]
166          endSquare = p.Rect(move.endcol*sq_size, move.endrow*sq_size, sq_size, sq_size)
167          p.draw.rect(screen, color, endSquare)
168          # draw captured piece onto rectangle
169          if move.piececaptured != "- -":
170              if move.isenpassantmove:
171                  enpassantRow = move.endrow + 1 if move.piececaptured[0] == 'b' else move.endrow - 1
172                  endSquare = p.Rect(move.endcol*sq_size, enpassantRow*sq_size, sq_size, sq_size)
173              screen.blit(images[move.piececaptured], endSquare)
174          # draw moving piece
175          screen.blit(images[move.piecemoved], p.Rect(c*sq_size, r*sq_size, sq_size, sq_size))
176          p.display.flip()
177          clock.tick(60)
```

Figure 19 : Helper Functions (animatemove)

This code from lines 152 - 177 is the helper function for animatemove. It animates the movement of a chess piece across the board as the program runs. Not only that, it also calculates intermediary positions of the piece as it moves and redraws the board and pieces with every frame.

```
179  #initialize global dictionary of images, will be called exactly once in the main
180  def loadimages():
181      pieces = ['wp', 'wR', 'wN', 'wB', 'wK', 'wQ', 'bp', 'bR', 'bN', 'bB', 'bK', 'bQ']
182      for piece in pieces:
183          images[piece] = p.image.load("chess/images/" + piece + ".png")
```

Figure 20 : Image Loading

This code form lines 180-183 is the image loading function, where it would load all the images for all the black and white pieces (w being white and b being black), from the image folder directory. The program would fetch the images, shown in line 183, from the directory "*chess/images*" + piece (where the name of the pieces is the same as the image file name) + "*.png*".

Drawing Functions

```
185     def drawGameState(screen, gs, selectedsquare, validmoves):
186         drawboard(screen)  # draw squares on the board
187         highlightSquares(screen, gs, selectedsquare, validmoves)  # highlight selected square and valid moves
188         drawpieces(screen, gs.board)  # draw pieces on top
189 
190     def highlightSquares(screen, gs, selectedsquare, validmoves):
191         if selectedsquare != ():
192             r, c = selectedsquare
193             # highlight selected square
194             s = p.Surface((sq_size, sq_size))
195             s.set_alpha(100)  # transparency value -> 0 transparent; 255 opaque
196             s.fill(p.Color('blue'))
197             screen.blit(s, (c*sq_size, r*sq_size))
198             # highlight valid moves
199             s.fill(p.Color('yellow'))
200             for move in validmoves:
201                 if move.startrow == r and move.startcol == c:
202                     screen.blit(s, (move.endcol*sq_size, move.endrow*sq_size))
203 
204     colors = [p.Color('white'), p.Color('gray')]
205     def drawboard(screen):
206         for r in range(dimension):
207             for c in range(dimension):
208                 color = colors[((r+c)%2)]
209                 p.draw.rect(screen, color, p.Rect(c*sq_size, r*sq_size, sq_size, sq_size))
210 
211     def drawpieces(screen, board):
212         for r in range(dimension):
213             for c in range(dimension):
214                 piece = board[r][c]
215                 if piece != "- -": #not empty squares
216                     screen.blit(images[piece], p.Rect(c*sq_size, r*sq_size, sq_size, sq_size))
```

Figure 21 : Full Drawing Functions

```
185     def drawGameState(screen, gs, selectedsquare, validmoves):
186         drawboard(screen)  # draw squares on the board
187         highlightSquares(screen, gs, selectedsquare, validmoves)  # highlight selected square and valid moves
188         drawpieces(screen, gs.board)  # draw pieces on top
```

Figure 22 : Drawing Function (drawGameState)

This code from lines 185 - 188 is the function where the code would call the helper functions to draw the board, highlight squares, and draw pieces.

```
190  def highlightSquares(screen, gs, selectedsquare, validmoves):
191      if selectedsquare != ():
192          r, c = selectedsquare
193          # highlight selected square
194          s = p.Surface((sq_size, sq_size))
195          s.set_alpha(100)  # transparency value -> 0 transparent; 255 opaque
196          s.fill(p.Color('blue'))
197          screen.blit(s, (c*sq_size, r*sq_size))
198          # highlight valid moves
199          s.fill(p.Color('yellow'))
200          for move in validmoves:
201              if move.startrow == r and move.startcol == c:
202                  screen.blit(s, (move.endcol*sq_size, move.endrow*sq_size))
```

Figure 23 : Highlight Squares Function

This code from lines 190 - 202 is behind the square highlighting feature, where when the user clicks on the piece that they want to move, the piece that they click would be highlighted blue, which can be seen in line 196 shown in *figure 24* below. Not only that, there are yellow highlights that show where the pieces can move. The pieces can only move to the squares that are highlighted in yellow, otherwise, the piece can't move there, since the move is also illegal in general chess rules.

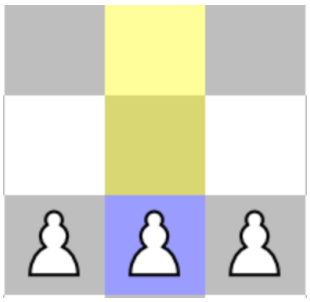


Figure 24 : Pawn Piece Highlight

```
204  colors = [p.Color('white'), p.Color('gray')]
205  def drawboard(screen):
206      for r in range(dimension):
207          for c in range(dimension):
208              color = colors[((r+c)%2)]
209              p.draw.rect(screen, color, p.Rect(c*sq_size, r*sq_size, sq_size, sq_size))
```

Figure 24 : Drawboard Function

This code from lines 204 - 209 shows the drawboard function, where it will draw the chessboard 'checkerboard' motif in the program with white and gray colors, shown in line 204.

```
211  def drawpieces(screen, board):
212      for r in range(dimension):
213          for c in range(dimension):
214              piece = board[r][c]
215              if piece != "- -": #not empty squares
216                  screen.blit(images[piece], p.Rect(c*sq_size, r*sq_size, sq_size, sq_size))
```

Figure 25 : Drawpieces Function

This code from lines 211 - 216 is used to draw all the individual pieces to the chessboard.

```
218  if __name__ == "__main__":
219      main()
```

Figure 26 : Program Execution

The last 2 lines, lines 218 - 219 is the standard code to execute the program.

ChessEngine.py

This file handles the overall logic of the program, moves, and rules of the pieces.

```
3  class GameState ():
4      def __init__(self):
5          self.board = [
6              ["bR", "bN", "bB", "bQ", "bK", "bB", "bN", "bR"],  #the order of the chessboard (1st row)
7              ["bp","bp","bp","bp","bp","bp","bp","bp"], #pawn row
8              ["- -","- -","- -","- -","- -","- -","- -","- -"], #blank space
9              ["- -","- -","- -","- -","- -","- -","- -","- -"],
10             ["- -","- -","- -","- -","- -","- -","- -","- -"],
11             ["- -","- -","- -","- -","- -","- -","- -","- -"],
12             ["wp","wp","wp","wp","wp","wp","wp","wp"], #pawn row (white)
13             ["wR", "wN", "wB", "wQ", "wK", "wB", "wN", "wR"],]
14         self.whiteToMove = True
15         self.moveLog = []
16         # Keep track of kings' positions
17         self.whiteKingLocation = (7, 4)
18         self.blackKingLocation = (0, 4)
19         self.checkmate = False
20         self.stalemate = False
21         # For castling
22         self.whiteCastleKingside = True
23         self.whiteCastleQueenside = True
24         self.blackCastleKingside = True
25         self.blackCastleQueenside = True
26         # For en passant
27         self.enpassantPossible = () # coordinates for the square where en passant capture is possible
```

Figure 27 : Full Game State Function

```
3  class GameState ():
4      def __init__(self):
5          self.board = [
6              ["bR", "bN", "bB", "bQ", "bK", "bB", "bN", "bR"],  #the order of the chessboard (1st row)
7              ["bp","bp","bp","bp","bp","bp","bp","bp"], #pawn row
8              ["- -","- -","- -","- -","- -","- -","- -","- -"], #blank space
9              ["- -","- -","- -","- -","- -","- -","- -","- -"],
10             ["- -","- -","- -","- -","- -","- -","- -","- -"],
11             ["- -","- -","- -","- -","- -","- -","- -","- -"],
12             ["wp","wp","wp","wp","wp","wp","wp","wp"], #pawn row (white)
13             ["wR", "wN", "wB", "wQ", "wK", "wB", "wN", "wR"],]
```

Figure 28 : Init Method Dictionary

This code from lines 3 - 13 opens the *GameState* class with an *init*, followed by a dictionary. It Initializes the chessboard as an 8x8 matrix, which can be seen in lines 6 - 13. Each string represents a piece ("*wK*" for white king, "*bp*" for black pawn, etc.) or an empty space ("- -"). This representation will be the first thing the user sees as they run the program, with this code as the default positions of the chessboard and pieces.

In Depth Game Logic

```
14             self.whiteToMove = True
15             self.moveLog = []
```

Figure 29 : General Game Logic

This code from line 14 - 15 shows the logic behind the game. It has a lot of functions, like indicating which player's turn it is. *True* means it's white's turn, *False* means it's black's turn, shown in line 14. It keeps a record of all moves made in the game for undoing moves or analyzing the game, shown in line 15.

```
16             # Keep track of kings' positions
17             self.whiteKingLocation = (7, 4)
18             self.blackKingLocation = (0, 4)
19             self.checkmate = False
20             self.stalemate = False
```

Figure 30 : Keeping Track of Kings' Positions

This code from lines 16 - 20 would keep track of the kings' position for both black and white pieces. Not only that, it also checks *checkmate* and *stalemate*, where the game would end then

self.checkmate / self.stalemate = True. *self.checkmate / self.stalemate = False* because initially when the game starts, the game has not ended yet.

```
21              # For castling
22              self.whiteCastleKingside = True
23              self.whiteCastleQueenside = True
24              self.blackCastleKingside = True
25              self.blackCastleQueenside = True
```

Figure 31 : Castling

This code from lines 21 - 25 is the code behind the move castling. Castling is a special move in chess where you do multiple unique actions. First of all, it is the only move where you may move two pieces in the same move. Castling is the only time in chess when it is legal to move the king more than one square as well. Not only that, but it is also the only move that develops your rook and protects your king (*chess.com*).

```
26              # For en passant
27              self.enpassantPossible = ()
```

Figure 32 : En Passant

This code, line 27 is the code behind the en passant move. En passant (French for "in passing") is a special chess rule allowing pawns to capture a pawn that has just passed it (*chess.com*). It allows the pawn piece to take away another pawn piece diagonally, where most of the time, the pawn only moves 1-2 boxes straight. It stores the square where an en passant capture is possible. An empty tuple means no en passant is available.

```
    def makemove(self, move):
        self.board[move.startrow][move.startcol] = "- -"
        self.board[move.endrow][move.endcol] = move.piecemoved
        self.moveLog.append(move) #log the move so can undo later
        self.whiteToMove = not self.whiteToMove #take turns

        # Update king's position
        if move.piecemoved == "wK":
            self.whiteKingLocation = (move.endrow, move.endcol)
        elif move.piecemoved == "bK":
            self.blackKingLocation = (move.endrow, move.endcol)

        # Pawn promotion
        if move.ispawnpromotion:
            promotedpiece = "Q" # auto-promote to queen for now
            self.board[move.endrow][move.endcol] = move.piecemoved[0] + promotedpiece

        # En passant
        if move.isenpassantmove:
            self.board[move.startrow][move.endcol] = "- -" # capturing the pawn

        # Update enpassantPossible
        if move.piecemoved[1] == "p" and abs(move.startrow - move.endrow) == 2:
            self.enpassantPossible = ((move.startrow + move.endrow)//2, move.startcol)
        else:
            self.enpassantPossible = ()

        # Castle move
        if move.iscastlemove:
            if move.endcol - move.startcol == 2: # Kingside castle
                self.board[move.endrow][move.endcol-1] = self.board[move.endrow][move.endcol+1]
                self.board[move.endrow][move.endcol+1] = "- -"
            else: # Queenside castle
                self.board[move.endrow][move.endcol+1] = self.board[move.endrow][move.endcol-2]
                self.board[move.endrow][move.endcol-2] = "- -"

        # Update castling rights
        self.updatecastlerights(move)

        # Check if a king was captured
        if move.piececaptured == "wK" or move.piececaptured == "bK":
            self.checkmate = True
```

Figure 33 : Full Makemove Function

```
29      def makemove(self, move):
30          self.board[move.startrow][move.startcol] = "- -"
31          self.board[move.endrow][move.endcol] = move.piecemoved
32          self.moveLog.append(move) #log the move so can undo later
33          self.whiteToMove = not self.whiteToMove #take turns
```

Figure 34 : Makemove Function

This code is the core code of the *makemove* function. It would move the piece to the target square and empties the starting square, shown in lines 30 - 31 and also logs the move and switches the turn to the other player, shown in lines 32 - 33.

```
35          # Update king's position
36          if move.piecemoved == "wK":
37              self.whiteKingLocation = (move.endrow, move.endcol)
38          elif move.piecemoved == "bK":
39              self.blackKingLocation = (move.endrow, move.endcol)
```

Figure 35 : Updating the King's Position

This code is the code to update the king's position when it's moved. The rest of the code (*figure 36*) is also pretty self explanatory, like in figure 36, the code is programmed to handle pawn promotion, shown in lines 42 - 44, automatically promoting to a queen. Lines 47 - 48 removes the pawn captured via en passant, lines 51 - 54 sets the square for en passant if a pawn moves two spaces forward, lines 57 - 63 handles castling by moving the rook to its correct position, and finally, line 66 updates the castling rights

```
41          # Pawn promotion
42          if move.ispawnpromotion:
43              promotedpiece = "Q" # auto-promote to queen for now
44              self.board[move.endrow][move.endcol] = move.piecemoved[0] + promotedpiece
45
46          # En passant
47          if move.isenpassantmove:
48              self.board[move.startrow][move.endcol] = "- -" # capturing the pawn
49
50          # Update enpassantPossible
51          if move.piecemoved[1] == "p" and abs(move.startrow - move.endrow) == 2:
52              self.enpassantPossible = ((move.startrow + move.endrow)//2, move.startcol)
53          else:
54              self.enpassantPossible = ()
55
56          # Castle move
57          if move.iscastlemove:
58              if move.endcol - move.startcol == 2: # Kingside castle
59                  self.board[move.endrow][move.endcol-1] = self.board[move.endrow][move.endcol+1]
60                  self.board[move.endrow][move.endcol+1] = "- -"
61              else: # Queenside castle
62                  self.board[move.endrow][move.endcol+1] = self.board[move.endrow][move.endcol-2]
63                  self.board[move.endrow][move.endcol-2] = "- -"
64
65          # Update castling rights
66          self.updatecastlerights(move)
```

Figure 36 : Updating Positions based on the move.

Undomove function

```
72      def undomove(self):
73          if len(self.moveLog) != 0:  # make sure there is a move to undo
74              move = self.moveLog.pop()
75              self.board[move.startrow][move.startcol] = move.piecemoved
76              self.board[move.endrow][move.endcol] = move.piececaptured
77              self.whiteToMove = not self.whiteToMove  # switch turns back
78
79              # Update king's position if needed
80              if move.piecemoved == "wK":
81                  self.whiteKingLocation = (move.startrow, move.startcol)
82              elif move.piecemoved == "bK":
83                  self.blackKingLocation = (move.startrow, move.startcol)
84
85              # Undo en passant move
86              if move.isenpassantmove:
87                  self.board[move.endrow][move.endcol] = "- -"  # leave landing square blank
88                  self.board[move.startrow][move.endcol] = move.piececaptured
89                  self.enpassantPossible = (move.endrow, move.endcol)
90
91              # Undo a 2 square pawn advance
92              if move.piecemoved[1] == "p" and abs(move.startrow - move.endrow) == 2:
93                  self.enpassantPossible = ()
```

Figure 37 : Full Undomove Function

This code, lines 72 - 93 is the *undomove* function. The function is to reverse the last move, which is a feature in the program when the user press ‘z’ to undo the move they just did. Lines 72 - 77 is programmed to restore the board state and switches the turn. It handles undoing special moves (en passant, pawn advance, castling, etc.) and resets flags like *self.enpassantPossible*. Lines 79 - 83 is responsible for updating the king's position on the chessboard after a move. It makes sure that the gamestate would accurately reflects the king's new location, which is critical for checking conditions like castling and avoiding illegal moves like for example, moving into check. Not only that, it would store the current position of the white and black kings. Lines 85 - 93 handles the en passant rule in chess, which allows a pawn to capture an opposing pawn that has moved two squares forward from its starting position. It sets or clears the self.enpassantPossible variable, which keeps track of the square where an en passant capture is possible.

```
95      def updatecastlerights(self, move):
96          if move.piecemoved == "wK":
97              self.whiteCastleKingside = False
98              self.whiteCastleQueenside = False
99          elif move.piecemoved == "bK":
100             self.blackCastleKingside = False
101             self.blackCastleQueenside = False
102         elif move.piecemoved == "wR":
103             if move.startrow == 7:
104                 if move.startcol == 0:
105                     self.whiteCastleQueenside = False
106                 elif move.startcol == 7:
107                     self.whiteCastleKingside = False
108         elif move.piecemoved == "bR":
109             if move.startrow == 0:
110                 if move.startcol == 0:
111                     self.blackCastleQueenside = False
112                 elif move.startcol == 7:
113                     self.blackCastleKingside = False
```

Figure 39 : Updating Castling Rights Function

This code is the core behind the castling moves. The code itself is pretty self explanatory, where in lines 96 - 98, if the white king is moved ("wK"), both castling rights for the white player (kingside and queenside) are forfeited, because castling requires the king to be unmoved. Similarly, if the black king is moved ("bK"), both castling rights for the black player are forfeited, shown in lines 99 - 101.

Lines 102 - 107 shows that if a white rook ("wR") moves, it would check if it started on row 7 (white's back rank). If it started in column 0 (queenside rook), queenside castling for white is forfeited. If it started in column 7 (kingside rook), kingside castling for white is forfeited. This also works the same for the black pieces shown in lines 108 - 113, If a black rook ("bR") moves, it would check if it started on row 0 (black's back rank). If it started in column 0 (queenside rook), queenside castling for black is forfeited and if it started in column 7 (kingside rook), kingside castling for black is forfeited.

```
115     def getvalidmoves(self):
116         """
117         Get all valid moves considering checks
118         """
119         temp_enpassant_possible = self.enpassantPossible
120         temp_castle_rights = CastleRights(self.whiteCastleKingside, self.blackCastleKingside,
121                                           self.whiteCastleQueenside, self.blackCastleQueenside)
122         # Get all possible moves
123         moves = self.getallpossiblemoves()
124         
125         # For each move, make it and see if it leaves king in check
126         for i in range(len(moves) - 1, -1, -1):
127             self.makemove(moves[i])
128             self.whiteToMove = not self.whiteToMove
129             if self.is_in_check():
130                 moves.remove(moves[i])
131             self.whiteToMove = not self.whiteToMove
132             self.undomove()
133 
134         # Restore the enpassant and castle rights
135         self.enpassantPossible = temp_enpassant_possible
136         self.whiteCastleKingside = temp_castle_rights.wks
137         self.blackCastleKingside = temp_castle_rights.bks
138         self.whiteCastleQueenside = temp_castle_rights.wqs
139         self.blackCastleQueenside = temp_castle_rights.bqs
140 
141         return moves
```

Figure 40 : Full Getvalidmoves Function

This code from lines 115 - 141 is the getvalidmoves function. It is programmed so that all moves returned are valid and do not leave the king in check. It does this by simulating each move (all the possibilities), verifying its legality, and restoring the gamestate afterward.

Lines 119 - 121 shows the current en passant possibility and castling rights are stored temporarily. This is necessary because these attributes might change during the process of simulating moves and need to be restored afterward.

```
115     def getvalidmoves(self):
116         """
117         Get all valid moves considering checks
118         """
119         temp_enpassant_possible = self.enpassantPossible
120         temp_castle_rights = CastleRights(self.whiteCastleKingside, self.blackCastleKingside,
121                                           self.whiteCastleQueenside, self.blackCastleQueenside)
122         # Get all possible moves
123         moves = self.getallpossiblemoves()
124 
125         # For each move, make it and see if it leaves king in check
126         for i in range(len(moves) - 1, -1, -1):
127             self.makemove(moves[i])
128             self.whiteToMove = not self.whiteToMove
129             if self.is_in_check():
130                 moves.remove(moves[i])
131             self.whiteToMove = not self.whiteToMove
132             self.undomove()
```

Figure 40 : Full Getvalidmoves Function

This code from lines 115 - 141 is the getvalidmoves function, which is programmed so that all moves returned are valid and do not leave the king in check. It does this by simulating each move, verifying its legality, and restoring the game state afterward. It has many features, like how the current en passant possibility and castling rights can be stored temporarily because these attributes might change during the process of simulating moves and need to be restored afterward shown in lines 119 - 120. It also retrieves a list of all possible moves without considering whether they leave the king in check, which includes moves that might not be valid according to the rules of chess (example : moves that leave the king in check), shown in line 123.

It has other features, like iterating through the list of moves backward to safely modify the list while iterating (avoiding index errors when removing elements). Line 127 - 128 is programmed so that the move is simulated using makemove, updating the game state as though the move were actually made. *self.whiteToMove* is toggled to switch to the opponent's perspective, as we need to check if the current player's king is in check after the move. It also checks for the king if it's in a checkmate position, as shown in lines 129 -130. Finally, lines 131 - 132 show that the move is undone using *undomove*, restoring the game state to what it was before the move was simulated, where *self.whiteToMove* is toggled back to the original player.

```
134             # Restore the enpassant and castle rights
135             self.enpassantPossible = temp_enpassant_possible
136             self.whiteCastleKingside = temp_castle_rights.wks
137             self.blackCastleKingside = temp_castle_rights.bks
138             self.whiteCastleQueenside = temp_castle_rights.wqs
139             self.blackCastleQueenside = temp_castle_rights.bqs
140
141             return moves
```

Figure 41 : Restore Temporary State

This code is programmed to restore the temporary state, where the en passant possibility and castling rights are restored to their original values. This will ensure that any temporary changes made during the simulation do not persist and will not affect future calculations. Then, it returns the filtered list of valid moves that comply with all chess rules, as shown in line 141.

```
143     def getallpossiblemoves(self):
144         moves = []  # start with an empty list instead of a hardcoded move
145         for r in range(len(self.board)):  # number of rows
146             for c in range(len(self.board[r])):  # number of columns in given row
147                 turn = self.board[r][c][0]
148                 if (turn == 'w' and self.whiteToMove) or (turn == 'b' and not self.whiteToMove):
149                     piece = self.board[r][c][1]
150                     if piece == 'p':
151                         self.getpawnmoves(r,c,moves)
152                     elif piece == 'R':
153                         self.getrookmoves(r,c,moves)
154                     elif piece == 'N':
155                         self.getknightmoves(r,c,moves)
156                     elif piece == 'B':
157                         self.getbishopmoves(r,c,moves)
158                     elif piece == 'Q':
159                         self.getqueenmoves(r,c,moves)
160                     elif piece == 'K':
161                         self.getkingmoves(r,c,moves)
162         return moves
```

Figure 42 : Full Getallpossiblemoves Function

```
143     def getallpossiblemoves(self):
144         moves = []  # start with an empty list instead of a hardcoded move
145         for r in range(len(self.board)):  # number of rows
146             for c in range(len(self.board[r])):  # number of columns in given row
147                 turn = self.board[r][c][0]
148                 if (turn == 'w' and self.whiteToMove) or (turn == 'b' and not self.whiteToMove):
149                     piece = self.board[r][c][1]
```

Figure 43 : Getallpossiblemoves Function

This code from lines 143 - 162 is the *getallpossiblemoves* function, which generates all potential moves for the current player, regardless of whether the moves are valid (example : ignoring whether they leave the king in check or not). It is designed to loop through every square on the board, identify the pieces belonging to the current player, and generate all their possible moves by calling piece-specific movement methods.

First, in line 144, it creates an empty list of moves, to store all possible moves for the current player. Each move is typically represented as an object or structure that tracks the start and end positions, the piece moved, and other relevant information. Next, in line 145, it loops through each row of the chessboard, where r represents the index of the current row (0 to 7 for an 8x8 chessboard). Similar to line 145, line 146 loops through each column within the current row, where c represents the index of the current column (0 to 7 for an 8x8 chessboard).

Lines 147 - 148 play a huge role in determining the piece's color. *self.board[r][c]* refers to the piece at position (r = row, c = column) on the board, represented as a string (example : ‘'wK' for the white king). It extracts the first character ('w' or 'b'), which represents the piece's color ('w' for white, 'b' for black). The condition checks if the piece belongs to the player whose turn it is (if it's white's turn (*self.whiteToMove* is True), only white pieces ('w') are considered and if it's black's turn (self.whiteToMove is False), only black pieces ('b') are considered).

Line 149 calls out the piece function, where piece extracts the second character of the string ('p', 'R', 'N', 'B', 'Q', or 'K' (p for pawn, R for rook and so on), which represents the type of the piece (since there are both black and white pieces).

Calling piece-specific move generators can be seen in lines 150 - 161, based on the type of the piece, it calls some methods to calculate its moves, like *self.getpawnmoves*(r, c, moves) generates all possible pawn moves from (r, c) and appends them to the moves list.
Similarly, *getrookmoves, getknightmoves, getbishopmoves, getqueenmoves,* and *getkingmoves* would do the same for their respective pieces. Each method determines valid moves for that specific piece type based on chess rules (example : pawns move forward, rooks move in straight lines, etc). Finally, line 162 returns the moves list, which now contains all potential moves for the current player.

```
    def getpawnmoves(self,r,c,moves):
        if self.whiteToMove:  # white pawn moves
            if r > 0 and self.board[r-1][c] == "- -":  # 1 square pawn advance
                moves.append(move((r,c), (r-1,c), self.board))
                if r == 6 and self.board[r-2][c] == "- -":  # 2 square pawn advance
                    moves.append(move((r,c), (r-2,c), self.board))
            # captures
            if r > 0 and c-1 >= 0 and self.board[r-1][c-1][0] == 'b':  # capture to left
                moves.append(move((r,c), (r-1,c-1), self.board))
            if r > 0 and c+1 <= 7 and self.board[r-1][c+1][0] == 'b':  # capture to right
                moves.append(move((r,c), (r-1,c+1), self.board))
            # En passant
            if (r-1, c-1) == self.enpassantPossible:
                attackingpiece = blockingpiece = False
                if self.whiteToMove:  # white to move
                    attackingpiece = self.board[r][c-1][0] == 'b'  # enemy piece to capture
                    blockingpiece = self.board[r-1][c-1] == "- -"  # blocking piece
                    if attackingpiece and blockingpiece:
                        m = move((r,c), (r-1,c-1), self.board)
                        m.isenpassantmove = True
                        moves.append(m)
            if (r-1, c+1) == self.enpassantPossible:
                attackingpiece = blockingpiece = False
                if self.whiteToMove:  # white to move
                    attackingpiece = self.board[r][c+1][0] == 'b'  # enemy piece to capture
                    blockingpiece = self.board[r-1][c+1] == "- -"  # blocking piece
                    if attackingpiece and blockingpiece:
                        m = move((r,c), (r-1,c+1), self.board)
                        m.isenpassantmove = True
                        moves.append(m)
        else:  # black pawn moves
            if r < 7 and self.board[r+1][c] == "- -":  # 1 square pawn advance
                moves.append(move((r,c), (r+1,c), self.board))
                if r == 1 and self.board[r+2][c] == "- -":  # 2 square pawn advance
                    moves.append(move((r,c), (r+2,c), self.board))
            # captures
            if r < 7 and c-1 >= 0 and self.board[r+1][c-1][0] == 'w':  # capture to left
                moves.append(move((r,c), (r+1,c-1), self.board))
            if r < 7 and c+1 <= 7 and self.board[r+1][c+1][0] == 'w':  # capture to right
                moves.append(move((r,c), (r+1,c+1), self.board))
            # En passant
            if (r+1, c-1) == self.enpassantPossible:
                attackingpiece = blockingpiece = False
                if not self.whiteToMove:  # black to move
                    attackingpiece = self.board[r][c-1][0] == 'w'  # enemy piece to capture
                    blockingpiece = self.board[r+1][c-1] == "- -"  # blocking piece
                    if attackingpiece and blockingpiece:
                        m = move((r,c), (r+1,c-1), self.board)
                        m.isenpassantmove = True
                        moves.append(m)
            if (r+1, c+1) == self.enpassantPossible:
                attackingpiece = blockingpiece = False
                if not self.whiteToMove:  # black to move
                    attackingpiece = self.board[r][c+1][0] == 'w'  # enemy piece to capture
                    blockingpiece = self.board[r+1][c+1] == "- -"  # blocking piece
                    if attackingpiece and blockingpiece:
                        m = move((r,c), (r+1,c+1), self.board)
                        m.isenpassantmove = True
                        moves.append(m)
```

Figure 44 : Full Getpawnmoves Function (White)

This code lines 165 - 224 the function *getpawnmoves*, which generates all possible moves for a pawn located at the position (r, c) on the chessboard and appends them to the moves list. It handles both

white and black pawns, considering their unique movement rules like advancing, capturing, and en passant. It will check if it's white's turn to move, and if the value is true, the logic for white pawns is executed, which can be seen from line 166. Single and double square advance moves can be seen from lines 167 - 170, where pawns can move one square forward if it's empty, ensure the pawn is not on the top row (*r > 0*), and check if the square directly above the pawn (*r-1, c*) is empty ("- -") and appends this move. On the other hand, white pawns can move two squares forward from their starting position (*r == 6)*, ensure the squares directly above (*r-1, c*) and two rows above (*r-2, c*) are empty, then execute the two-square advance move for double square moves. Capturing the pieces from left to right can also be seen from lines 172 - 175, where it checks if the pawn can capture diagonally to the left and then, it will make sure that the target square is within bounds (*c-1 >= 0*), and if the square contains an opponent's piece ('b' for Black) it will execute the capture move. It does the same to capture to the right, which can be seen from lines 174 - 175.

Next, en passant movements can be seen in lines 177 - 194 for capturing pieces on the left and right. It checks if the target square (*r-1, c-1*) m.isenpassantmoveassantPossible (the square where en passant is valid). Ensures the pawn captures an opponent's pawn (*attackingpiece*) and that the square in front of it is empty (*blockingpiece*). Marks the move as an en passant move (*m.isenpassantmove = True*), seen in line 184, and executes it. It also works the same for the en passant capture to the right,

```
196             else:  # black pawn moves
197                 if r < 7 and self.board[r+1][c] == "- -":  # 1 square pawn advance
198                     moves.append(move((r,c), (r+1,c), self.board))
199                     if r == 1 and self.board[r+2][c] == "- -":  # 2 square pawn advance
200                         moves.append(move((r,c), (r+2,c), self.board))
201                 # captures
202                 if r < 7 and c-1 >= 0 and self.board[r+1][c-1][0] == 'w':  # capture to left
203                     moves.append(move((r,c), (r+1,c-1), self.board))
204                 if r < 7 and c+1 <= 7 and self.board[r+1][c+1][0] == 'w':  # capture to right
205                     moves.append(move((r,c), (r+1,c+1), self.board))
206                 # En passant
207                 if (r+1, c-1) == self.enpassantPossible:
208                     attackingpiece = blockingpiece = False
209                     if not self.whiteToMove:  # black to move
210                         attackingpiece = self.board[r][c-1][0] == 'w'  # enemy piece to capture
211                         blockingpiece = self.board[r+1][c-1] == "- -"  # blocking piece
212                         if attackingpiece and blockingpiece:
213                             m = move((r,c), (r+1,c-1), self.board)
214                             m.isenpassantmove = True
215                             moves.append(m)
216                 if (r+1, c+1) == self.enpassantPossible:
217                     attackingpiece = blockingpiece = False
218                     if not self.whiteToMove:  # black to move
219                         attackingpiece = self.board[r][c+1][0] == 'w'  # enemy piece to capture
220                         blockingpiece = self.board[r+1][c+1] == "- -"  # blocking piece
221                         if attackingpiece and blockingpiece:
222                             m = move((r,c), (r+1,c+1), self.board)
223                             m.isenpassantmove = True
224                             moves.append(m)
```

Figure 45 : Black Pawn Moves

Similar to the white pawn moves, lines 196 - 224 also does the same en passant captures like the white one in figure 46, but this is the code for the black pieces.

```
226     def getrookmoves(self,r,c,moves):
227         directions = [(-1,0), (0,-1), (1,0), (0,1)]  # up, left, down, right
228         enemy_color = 'b' if self.whiteToMove else 'w'
229 
230         for d in directions:
231             for i in range(1,8):
232                 endrow = r + d[0] * i
233                 endcol = c + d[1] * i
234                 if 0 <= endrow < 8 and 0 <= endcol < 8:  # check if on board
235                     endpiece = self.board[endrow][endcol]
236                     if endpiece == "- -":  # empty space is valid
237                         moves.append(move((r,c), (endrow,endcol), self.board))
238                     elif endpiece[0] == enemy_color:  # capture enemy piece
239                         moves.append(move((r,c), (endrow,endcol), self.board))
240                         break
241                     else:  # friendly piece
242                         break
243                 else:  # off board
244                     break
```

Figure 46 : Getrookmoves Function

The *getrookmoves* function from lines 226 - 244 generates all possible moves for a rook located at (r, c) by iterating through its four movement directions (up, left, down, right) as defined in the directions list. First, it determines the opponent's color using *enemy_color*, based on whose turn it is. For each direction, the function iterates through the board, incrementing the row and column indices (*endrow, endcol*), step by step. It checks if the current square is within board boundaries (*0 <= endrow < 8 and 0 <= endcol < 8*) shown in line 234. If the square is empty (*endpiece == "- -"*), shown in line 236, it appends the move to the moves list. If the square contains an enemy piece (*endpiece[0] == enemy_color*) shown in line 238, it appends the capturing move and stops further exploration in that direction. If the square contains a friendly piece or is out of bounds, the loop for that direction stops. This will make sure that there is accurate possibilities of rook moves while considering board boundaries, empty spaces, and collisions with friendly or enemy pieces.

```
246     def getknightmoves(self,r,c,moves):
247         knight_moves = [(-2,-1), (-2,1), (-1,-2), (-1,2),
248                         (1,-2), (1,2), (2,-1), (2,1)]
249         ally_color = 'w' if self.whiteToMove else 'b'
250 
251         for m in knight_moves:
252             endrow = r + m[0]
253             endcol = c + m[1]
254             if 0 <= endrow < 8 and 0 <= endcol < 8:
255                 endpiece = self.board[endrow][endcol]
256                 if endpiece[0] != ally_color:  # not an ally piece (empty or enemy)
257                     moves.append(move((r,c), (endrow,endcol), self.board))
```

Figure 47 : Getknightmoves Function

The *getknightmoves* function shown in lines 246 - 257 calculates all valid moves for a knight positioned at (r, c) by iterating through its potential moves, which can be seen in line 247, in the *knight_moves* list as relative offsets. It first determines the allied color (w for white or b for black) based on the current turn. For each possible knight move, the potential destination square (endrow, endcol) is calculated by adding the offset to the knight's current position. It will make sure that the destination is within the board boundaries (*0 <= endrow < 8 and 0 <= endcol < 8*) shown in line 254. If the destination square is either empty or occupied by an enemy piece (*endpiece[0] != ally_color*) shown in line 256, it appends the move to the moves list by creating a new move object. This logic ensures that knights can move to all valid positions, capturing enemy pieces while avoiding friendly pieces or out of bounds moves.

```
259     def getbishopmoves(self,r,c,moves):
260         directions = [(-1,-1), (-1,1), (1,-1), (1,1)]  # diagonals
261         enemy_color = 'b' if self.whiteToMove else 'w'
262 
263         for d in directions:
264             for i in range(1,8):
265                 endrow = r + d[0] * i
266                 endcol = c + d[1] * i
267                 if 0 <= endrow < 8 and 0 <= endcol < 8:
268                     endpiece = self.board[endrow][endcol]
269                     if endpiece == "- -":
270                         moves.append(move((r,c), (endrow,endcol), self.board))
271                     elif endpiece[0] == enemy_color:
272                         moves.append(move((r,c), (endrow,endcol), self.board))
273                         break
274                     else:
275                         break
276                 else:
277                     break
```

Figure 48 : Getbishopmoves

The *getbishopmoves* function generates all valid moves for a bishop located at (r, c) by exploring its movement along the four diagonal directions, which is defined in the directions list. The function first determines the *enemy_color* , in line 261 based on the current turn (b for black if white is to move, and vice versa). For each diagonal direction, it iterates through possible moves, incrementing the step size i from 1 to 7 to cover all squares along the direction. It calculates the target square (endrow, endcol) using the direction vector scaled by i. If the target square is within the board bounds (*0 <= endrow < 8 and 0 <= endcol < 8*), it checks the contents of the square (endpiece) shown in line 267. If the square is empty ("- -") or contains an enemy piece (*endpiece[0] == enemy_color*), it adds the move to the moves list, stopping further exploration in the current direction after capturing an enemy. If the square contains a friendly piece or is out of bounds, it breaks out of the loop, just to make sure that bishops don't jump over pieces or move outside the board.

```
279         def getqueenmoves(self,r,c,moves):
280             self.getrookmoves(r,c,moves)  # Queen combines rook
281             self.getbishopmoves(r,c,moves)  # and bishop moves
```

Figure 49 : Getqueenmoves Function

The *getqueenmoves* function generates all valid moves for a queen located at (r, c) by leveraging the movement patterns of both rooks and bishops, as the queen can move like both. The first line calls the getrookmoves function, which calculates the queen's valid moves along vertical and horizontal directions (like a rook). The second line calls the getbishopmoves function, which generates the queen's valid moves along the diagonal directions (like a bishop). Both functions append their respective valid moves to the moves list passed as an argument. Since the queen's movement is a combination of rook and bishop moves, this implementation avoids duplication and ensures the queen's full range of motion is accounted for efficiently.

```
283         def getkingmoves(self,r,c,moves):
284             king_moves = [(-1,-1), (-1,0), (-1,1), (0,-1),
285                           (0,1), (1,-1), (1,0), (1,1)]
286             ally_color = 'w' if self.whiteToMove else 'b'
```

Figure 50 : Getkingmoves Function

The *getkingmoves* function begins by defining the possible directions the king can move, stored in the *king_moves* list. Each tuple in the list represents a relative offset in rows and columns for a single step in all eight possible directions (diagonally, vertically, and horizontally). The function then determines the color of the king's allied pieces by assigning 'w' (white) if it is white's turn to move or 'b' (black) if it's their turn. This information is stored in the variable *ally_color* and is used later to make sure that the king

does not move to a square occupied by an allied piece. These preparatory steps set the stage for iterating through the possible moves and validating their legality.

```
288             for i in range(8):
289                 endrow = r + king_moves[i][0]
290                 endcol = c + king_moves[i][1]
291                 if 0 <= endrow < 8 and 0 <= endcol < 8:
292                     endpiece = self.board[endrow][endcol]
293                     if endpiece[0] != ally_color:  # not an ally piece
294                         # Check for castling
295                         if endpiece == "- -" and abs(endcol - c) == 2:
296                             if endcol - c == 2: # Kingside castle
297                                 if self.whiteToMove and self.whiteCastleKingside:
298                                     moves.append(move((r,c), (endrow,endcol), self.board))
299                                 elif not self.whiteToMove and self.blackCastleKingside:
300                                     moves.append(move((r,c), (endrow,endcol), self.board))
301                             else: # Queenside castle
302                                 if self.whiteToMove and self.whiteCastleQueenside:
303                                     moves.append(move((r,c), (endrow,endcol), self.board))
304                                 elif not self.whiteToMove and self.blackCastleQueenside:
305                                     moves.append(move((r,c), (endrow,endcol), self.board))
306                         else:
307                             moves.append(move((r,c), (endrow,endcol), self.board))
```

Figure 51 : King Legal Moves

This code from lines 288 - 307 generates all legal moves for a king, including normal moves, captures, and castling, while making sure that the king does not move into an ally-occupied square. The for loop iterates through all eight possible king moves, calculating the potential destination square (*endrow, endcol)* by adding the respective row and column offsets from the *king_moves* list. It checks if the destination square lies within the board's bounds (*0 <= endrow < 8 and 0 <= endcol < 8*). If valid, the piece at the destination (*endpiece*) is retrieved. If the destination square is not occupied by an allied piece (*endpiece[0] != ally_color*), as seen in line 293, it is further examined for special castling conditions. If the destination square is empty (*endpiece == "- -"*) and involves moving two columns horizontally (*abs(endcol - c) == 2*), it checks whether the move is a kingside or queenside castle. For kingside castling, it verifies the player's turn and castling rights (*self.whiteCastleKingside or self.blackCastleKingside)* and appends the move if valid. However, for queenside castling, the corresponding castling rights (*self.whiteCastleQueenside or self.blackCastleQueenside*) are checked before the move is added. If the move is not a castling attempt, it simply appends the valid move to the moves list, whether it is a normal move or a capture.

```
309     def is_in_check(self):
310         """
311         Determine if the current player is in check
312         """
313         if self.whiteToMove:
314             return self.square_under_attack(self.whiteKingLocation[0], self.whiteKingLocation[1])
315         else:
316             return self.square_under_attack(self.blackKingLocation[0], self.blackKingLocation[1])
```

Figure 52 : King (In Check) Indication

This function from lines 309 - 316 checks if the current player's king is under attack, indicating a check situation. The function begins by checking whose turn it is using the *self.whiteToMove flag*. If *self.whiteToMove* is True, it retrieves the current location of the white king from *self.whiteKingLocation* (a tuple containing the row and column) and calls the helper method *square_under_attack(),* shown in line 314 to determine if that square is under attack by any opposing piece, then the result of this check is returned. If *self.whiteToMove* is False, it instead retrieves the black king's position from *self.blackKingLocation* and performs the same check with *square_under_attack()*. This method shows the logic for determining if the current player's king is in a threatened position, returning True if the king is under attack and False otherwise.

```
318     def square_under_attack(self, r, c):
319         """
320         Determine if the square (r, c) is under attack by enemy pieces
321         """
322         self.whiteToMove = not self.whiteToMove  # Switch to opponent's perspective
323         opp_moves = self.getallpossiblemoves()
324         self.whiteToMove = not self.whiteToMove  # Switch back
325         for move in opp_moves:
326             if move.endrow == r and move.endcol == c:  # Square is under attack
327                 return True
328         return False
```

Figure 53 : Piece Under Attack Function

This function from lines 318 - 328 checks if a specific square (r, c) is under attack by any enemy piece. The function first flips the *self.whiteToMove* flag to simulate the opponent's turn, allowing the generation of all possible moves for the enemy using the *getallpossiblemoves()* method. After generating the opponent's moves, it restores the *self.whiteToMove* flag to its original state to maintain the game state. It then iterates over the opponent's moves stored in *opp_moves*. For each move, it checks if the ending position of the move corresponds to the square (r, c) being analyzed. If any move ends at (r, c), the square is confirmed to be under attack, and the function immediately returns True. If no such move exists, the function returns False, indicating the square is safe. This ensures that the function can verify threats to a specific square efficiently while preserving the game state.

```
330     def check_checkmate_stalemate(self):
331         """
332         Check if the current position is checkmate or stalemate
333         """
334         if len(self.getvalidmoves()) == 0:  # No valid moves
335             if self.is_in_check():
336                 self.checkmate = True
337             else:
338                 self.stalemate = True
339         else:
340             self.checkmate = False
341             self.stalemate = False
```

Figure 54 : Checkmate / Stalemate Function

This function determines whether the current game state is a checkmate or stalemate. The function begins by checking if there are no valid moves available for the current player using *len(self.getvalidmoves()) == 0* seen in line 334. If there are no valid moves, it checks whether the current player is in check using the *self.is_in_check()* function. If the player is in check, it sets *self.checkmate* to True, indicating that the game is a checkmate scenario. Otherwise, it sets *self.stalemate* to True, showing that the game is in a stalemate. If valid moves do exist, both *self.checkmate* and *self.stalemate* are set to False, indicating that the game continues. This function ensures proper recognition of endgame conditions by combining the absence of valid moves and the king's safety status.

```
343 class CastleRights():
344     def __init__(self, wks, bks, wqs, bqs):
345         self.wks = wks
346         self.bks = bks
347         self.wqs = wqs
348         self.bqs = bqs
```

Figure 55 : Castlerights Function

This class, castlerights from lines 343 - 348 shapes the rights for castling in a chess game for both players. The *__init__* method initializes an instance of the *CastleRights* class with four attributes (wks (white king-side castle), bks (black king-side castle), wqs (white queen-side castle), and bqs (black queen-side castle)), shown in line 344. Each parameter represents whether the respective player has the right to castle on the specified side. The values for these attributes are provided as arguments during the

object creation. This class also serves as a simple container for managing and tracking castling rights throughout the game.

```
350  class move():
351      #maps keys to values where key : value
352      #align the chess coordinates with the python based coordinates (0,0 in py is a8 in chess) goes down by matrix
353      rankstorows = {"1" : 7, "2" : 6, "3" : 5, "4" : 4, "5" : 3, "6" : 2, "7" : 1, "8" : 0}
354      rowstoranks = {v : k for k, v in rankstorows.items()} #reverse vk kv
355      filestocols = {"a" : 0, "b" : 1, "c" : 2, "d" : 3, "e" : 4, "f" : 5, "g" : 6, "h" : 7}
356      colstofiles = {v : k for k, v in filestocols.items()}
```

Figure 56 : Move Class

This *move* class from lines 350 - 356, defines mappings between chess coordinates (like "a1", "b3") and Python-based matrix-style coordinates (like (0, 0), (2, 3)) to view easier chess move handling and processing. The *rankstorows* dictionary maps chess ranks (1 to 8) to Python-style rows (7 to 0), aligning the chess board's vertical coordinates with how they are represented in a list or 2D array in Python. The *rowstoranks* dictionary reverses this mapping, mapping Python-style rows (0 to 7) back to the corresponding chess ranks (1 to 8). The *filestocols* dictionary in line 355 maps chess files (a to h) to Python-style columns (0 to 7), aligning horizontal chess coordinates with Python list indexing. Finally, the *colstofiles* dictionary reverses this, mapping Python-style columns (0 to 7) back to the corresponding chess files (a to h). These mappings make it easier to manipulate and convert between chess coordinates and matrix-based coordinates in the program.

```
358      def __init__(self, startsq, endsq, board):
359          self.startrow = startsq[0] #move from startcol0 to start col1 (pieces moving)
360          self.startcol = startsq[1]
361          self.endrow = endsq[0]
362          self.endcol = endsq[1]
363          self.piecemoved = board[self.startrow][self.startcol]
364          self.piececaptured = board[self.endrow][self.endcol] #information of all a move in 1 place
365          # Pawn promotion
366          self.ispawnpromotion = (self.piecemoved == "wp" and self.endrow == 0) or \
367                                 (self.piecemoved == "bp" and self.endrow == 7)
368          # En passant
369          self.isenpassantmove = False
370          # Castle move
371          self.iscastlemove = False
372          self.moveid = self.startrow * 1000 + self.startcol * 100 + self.endrow * 10 + self.endcol #move tht number into the thousand
373          print(self.moveid)
```

Figure 57 : Init Function (Move Class)

This *__init__* function in the move class initializes the details of a chess move, including the starting and ending positions of a piece, the piece moved and captured, and special move details like pawn promotion, en passant, and castling. The startrow and startcol attributes store the starting row and column of the move, while endrow and endcol store the ending position. The *piecemoved* attribute in line 363 stores the piece being moved (example : "wp" for white pawn), and *piececaptured* stores the piece that was captured (if any). The *ispawnpromotion* flag is set to True if the move involves a pawn reaching the promotion rank (row 0 for white or row 7 for black). The *isenpassantmove* flag is initialized as False, which will be updated later if the move is an en passant capture. Similarly, the *iscastlemove* flag is initialized as False for regular moves and will be updated if the move is a castling move. The *moveid* attribute is assigned a unique ID based on the start and end coordinates, which helps to identify and track

the move in the game. Finally, the *print(self.moveid)* prints the moveid for debugging or tracking purposes.

```
375         #overriding the equals method
376         def __eq__(self, other):
377             if isinstance(other, move):
378                 return self.moveid == other.moveid
379             return False
```

Figure 58 : Overriding Equals Method

This *__eq__* function defines how to compare two move objects for equality. The method starts by checking if the other object is an instance of the move class using *isinstance(other, move*). If other is indeed a move object, the function compares their moveid attributes. Since mmoveidis unique for each move, if both move objects have the same moveid, they are considered equal, and the method returns True. If other is not a move object, the method returns False, indicating the objects are not equal. This method allows for direct comparisons of move objects, ensuring that two moves are considered equal only if they represent the same move (i.e., have the same start and end positions and same unique moveid).

```
382     def getchessnotation(self) :
383         #making real chess notations
384         return self.getrankfile(self.startrow, self.startcol) + self.getrankfile(self.endrow, self.endcol)
```

Figure 59 : Getchessnotation Function

The getchessnotation method generates the standard chess notation for a move. The method first calls self.getrankfile(self.startrow, self.startcol) to convert the starting position of the move (defined by startrow and startcol) into chessboard coordinates (like "e2"). It then calls self.getrankfile(self.endrow, self.endcol) to convert the ending position (defined by endrow and endcol) into the corresponding chess notation (like "e4"). The method concatenates these two results, combining the starting and ending positions into a single string representing the move in standard algebraic chess notation, such as "e2e4". This method is useful for displaying moves in a human-readable format on the chessboard.

```
386         def getrankfile(self, r, c) : #row column
387             return self.colstofiles[c] + self.rowstoranks[r]
```

Figure 60 : Getrankfile Function

The getrankfile method converts a given row and column on the board to standard chess notation. It takes two arguments, r (row) and c (column), which represent the position on the chessboard. The method first uses self.colstofiles[c] to convert the column index c into a letter (like "a", "b", "c", etc.) based on the colstofiles dictionary. Then, it uses self.rowstoranks[r] to convert the row index r into the corresponding rank number (like "1", "2", "3", etc.) based on the rowstoranks dictionary. The method

concatenates the letter and number to return the chessboard coordinate as a string, such as "e4" or "a7". This is a common way to represent positions on a chessboard.

ChessAI.py

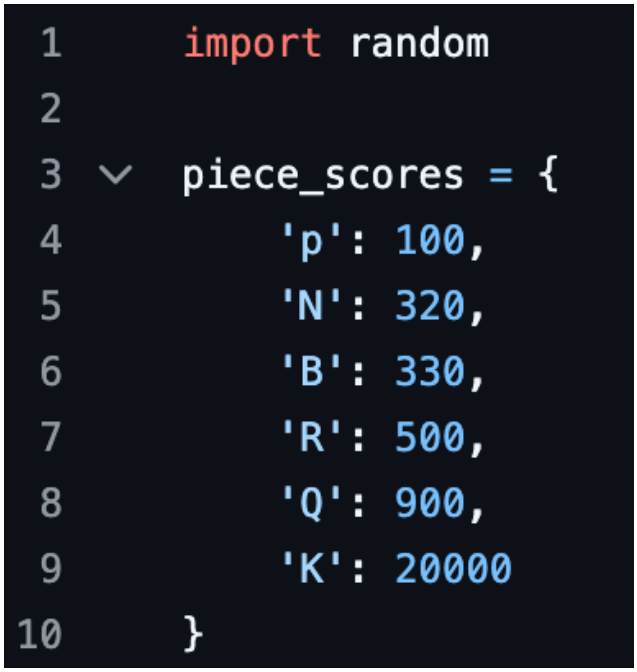
This file is different from the other 2 files, but this file is the logic behind the opponent's (AI) moves including board evaluation, move, and overall decision-making. The logic behind the AI is made with an algorithm called the Minimax Algorithm with Alpha-Beta Pruning.

About Minimax Algorithm with Alpha-Beta Pruning

The Minimax algorithm is a decision-making strategy used in two-player, zero-sum games, where one player's gain is the other player's loss. It works by simulating all possible moves and their outcomes, evaluating them recursively using a tree structure. The algorithm alternates between the two players, maximizing the player's score and minimizing the opponent's score at each level. Alpha-Beta Pruning is an optimization technique for Minimax that reduces the number of nodes evaluated by the algorithm. It maintains two values, alpha (the best score the maximizing player can guarantee) and beta (the best score the minimizing player can guarantee), and prunes branches of the tree that cannot influence the final decision, thus speeding up the search process.

In a chess game implemented with Pygame, the Minimax algorithm with Alpha-Beta Pruning can significantly improve the AI's decision-making and performance. Without pruning, the AI would need to evaluate every possible move in the game tree, which can be computationally expensive as the number of moves increases, especially in complex games like chess. With Alpha-Beta Pruning, redundant branches of the tree are eliminated, reducing the number of nodes the algorithm needs to explore. This allows the AI to make decisions faster, even in real-time scenarios, and respond with more intelligent moves. In Pygame, this makes the game more challenging for players by enabling the AI to evaluate positions more efficiently without sacrificing the quality of the gameplay experience. The pruning allows for deeper searches in the same amount of time, making the AI more strategic.

Code Analysis



```
import random

piece_scores = {
    'p': 100,
    'N': 320,
    'B': 330,
    'R': 500,
    'Q': 900,
    'K': 20000
}
```

Figure 61 : Import Random

This code defines a dictionary called piece_scores that assigns a numeric value to each type of chess piece. Each key in the dictionary corresponds to a specific piece: 'p' for pawns, 'N' for knights, 'B' for bishops, 'R' for rooks, 'Q' for queens, and 'K' for kings. The associated values represent the relative strength of each piece in terms of its value in the game. The values are based on commonly accepted approximations of piece worth in chess, with pawns assigned 100 points, knights 320 points, bishops 330 points, rooks 500 points, queens 900 points, and kings a high value of 20,000 points, reflecting their importance in the game (particularly for the king's survival). This scoring system can be used in evaluation functions for chess engines to assess the strength of positions or moves.

```
# Piece position tables - higher values for better positions
pawn_table = [
    [ 0,  0,  0,  0,  0,  0,  0,  0],
    [50, 50, 50, 50, 50, 50, 50, 50],
    [10, 10, 20, 30, 30, 20, 10, 10],
    [ 5,  5, 10, 25, 25, 10,  5,  5],
    [ 0,  0,  0, 20, 20,  0,  0,  0],
    [ 5, -5,-10,  0,  0,-10, -5,  5],
    [ 5, 10, 10,-20,-20, 10, 10,  5],
    [ 0,  0,  0,  0,  0,  0,  0,  0]
]
```

Figure 62 : Piece Position Table (Pawn)

This code defines a pawn_table, which is a position evaluation table for pawns in a chess game. The table is a list of lists representing the value of a pawn on each square of the chessboard. The rows correspond to the ranks (1 to 8), and the columns correspond to the files (a to h). The numbers within the table indicate the relative value of placing a pawn on a specific square, with higher values indicating more favorable positions. The first row ([0, 0, 0, 0, 0, 0, 0, 0]) corresponds to the back rank, where pawns are least valuable because they are just starting the game. The second row ([50, 50, 50, 50, 50, 50, 50, 50]) shows that pawns are more valuable when they are on the second rank, reflecting their ability to move forward and promote.

As you move down the table, the values adjust to reflect the increasing importance of controlling central and advanced squares, with the middle rows (3rd to 6th) featuring higher values for pawns in central squares (like d4, e4, d5, e5). The lower rows (7th and 8th) show negative values as pawns reach the end of their journey, where they become less valuable unless they are close to promotion. This table helps a chess engine evaluate pawn positions in terms of how advantageous they are, considering factors like central control and pawn structure. It is commonly used in chess engines for positional evaluation.

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

bishop_table = [
    [-20,-10,-10,-10,-10,-10,-10,-20],
    [-10,  0,  0,  0,  0,  0,  0,-10],
    [-10,  0,  5, 10, 10,  5,  0,-10],
    [-10,  5,  5, 10, 10,  5,  5,-10],
    [-10,  0, 10, 10, 10, 10,  0,-10],
    [-10, 10, 10, 10, 10, 10, 10,-10],
    [-10,  5,  0,  0,  0,  0,  5,-10],
    [-20,-10,-10,-10,-10,-10,-10,-20]
]
```

```
rook_table = [
    [ 0,  0,  0,  0,  0,  0,  0,  0],
    [ 5, 10, 10, 10, 10, 10, 10,  5],
    [-5,  0,  0,  0,  0,  0,  0, -5],
    [-5,  0,  0,  0,  0,  0,  0, -5],
    [-5,  0,  0,  0,  0,  0,  0, -5],
    [-5,  0,  0,  0,  0,  0,  0, -5],
    [-5,  0,  0,  0,  0,  0,  0, -5],
    [ 0,  0,  0,  5,  5,  0,  0,  0]
]

queen_table = [
    [-20,-10,-10, -5, -5,-10,-10,-20],
    [-10,  0,  0,  0,  0,  0,  0,-10],
    [-10,  0,  5,  5,  5,  5,  0,-10],
    [ -5,  0,  5,  5,  5,  5,  0, -5],
    [  0,  0,  5,  5,  5,  5,  0, -5],
    [-10,  5,  5,  5,  5,  5,  0,-10],
    [-10,  0,  5,  0,  0,  0,  0,-10],
    [-20,-10,-10, -5, -5,-10,-10,-20]
]
```

```
king_table = [
    [-30,-40,-40,-50,-50,-40,-40,-30],
    [-30,-40,-40,-50,-50,-40,-40,-30],
    [-30,-40,-40,-50,-50,-40,-40,-30],
    [-30,-40,-40,-50,-50,-40,-40,-30],
    [-20,-30,-30,-40,-40,-30,-30,-20],
    [-10,-20,-20,-20,-20,-20,-20,-10],
    [ 20, 20,  0,  0,  0,  0, 20, 20],
    [ 20, 30, 10,  0,  0, 10, 30, 20]
]
```

Table 2 : Knight, Bishop, Rook, Queen and King Table

```
piece_position_scores = {
        'p': pawn_table,
        'N': knight_table,
        'B': bishop_table,
        'R': rook_table,
        'Q': queen_table,
        'K': king_table
    }
```

Figure 63 : Piece Position Scores

```python
def get_position_score(piece, row, col, is_white):
    if piece == "- -":
        return 0

    piece_type = piece[1]
    if piece_type not in piece_position_scores:
        return 0

    position_table = piece_position_scores[piece_type]

    # For black pieces, we flip the table
    if not is_white:
        row = 7 - row

    return position_table[row][col]
```

Figure 64 : Getpositionscore Function

The get_position_score function calculates the positional score of a chess piece based on its type and location on the board. It first checks if the square is empty and returns a score of 0 if so. The piece's type is then extracted, and if its type has an associated position table in piece_position_scores, the function retrieves the table. If the piece is black, the row is flipped to account for the mirrored perspective of black pieces. Finally, the function returns the score from the appropriate position table at the given row and column. This function helps evaluate the strategic value of a piece's position on the chessboard, considering both the type of piece and its location.

```python
def evaluate_board(board):
    """Evaluates the board position. Positive score favors white, negative favors black."""
    score = 0

    # Material and position score
    for row in range(8):
        for col in range(8):
            piece = board[row][col]
            if piece != "- -":
                # Material score
                piece_value = piece_scores.get(piece[1], 0)
                if piece[0] == 'w':
                    score += piece_value
                    score += get_position_score(piece, row, col, True)
                else:
                    score -= piece_value
                    score -= get_position_score(piece, row, col, False)

    return score
```

Figure 65 : Evaluate Board Function

The evaluate_board function calculates the overall evaluation score of a given chessboard position, where a positive score favors White and a negative score favors Black. It initializes a variable score to 0, representing the net evaluation. The function then iterates over each square on the 8x8 chessboard, checking each piece. If the square is not empty ("- -"), it retrieves the piece's material value from piece_scores based on the piece's type (e.g., pawn, knight). Depending on whether the piece belongs to White or Black, the score is adjusted: for White, it adds both the material score and the positional score (calculated using the get_position_score function); for Black, it subtracts both the material and positional scores. After evaluating all pieces on the board, the final score is returned, representing the board's favorability for White or Black. This evaluation considers both material advantage and strategic positioning on the board.

```python
def minimax(gs, depth, alpha, beta, is_maximizing):
    """
    Minimax algorithm with alpha-beta pruning for chess AI
    gs: GameState object
    depth: How many moves to look ahead
    alpha, beta: Parameters for alpha-beta pruning
    is_maximizing: True if it's white's turn, False if black's turn
    """
    if depth == 0:
        return evaluate_board(gs.board)

    valid_moves = gs.getvalidmoves()

    if is_maximizing:
        max_eval = float('-inf')
        for move in valid_moves:
            gs.makemove(move)
            eval = minimax(gs, depth - 1, alpha, beta, False)
            gs.undomove()
            max_eval = max(max_eval, eval)
            alpha = max(alpha, eval)
            if beta <= alpha:
                break
        return max_eval
    else:
        min_eval = float('inf')
        for move in valid_moves:
            gs.makemove(move)
            eval = minimax(gs, depth - 1, alpha, beta, True)
            gs.undomove()
            min_eval = min(min_eval, eval)
            beta = min(beta, eval)
            if beta <= alpha:
                break
        return min_eval
```

Figure 66 : Minimax Function

The minimax function implements the Minimax algorithm with alpha-beta pruning, which is used to make optimal decisions in a game like chess by simulating all possible moves up to a given depth. The function takes the following parameters: gs is the current GameState object, depth represents how many moves ahead the algorithm should look, alpha and beta are parameters used for pruning, and is_maximizing indicates whether it's White's turn (True) or Black's turn (False). The base case is when depth reaches 0, at which point the function returns the evaluation score of the current board position using evaluate_board. Then, the function calculates all valid moves for the current player. If it's the maximizing player's turn (White), it initializes max_eval to negative infinity and iterates over each valid move, making the move, calling minimax recursively to evaluate the resulting position, and then undoing

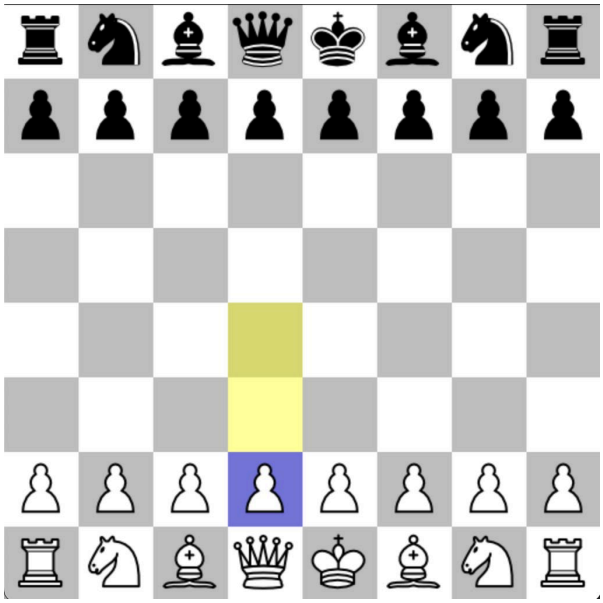
the move. It updates max_eval with the maximum evaluation found and adjusts alpha to the maximum of alpha and the current evaluation. If beta is less than or equal to alpha, pruning occurs (example : the loop breaks early because further exploration is unnecessary). On the other hand, if it's the minimizing player's turn (Black), the function does the same but with min_eval initialized to positive infinity and updates beta accordingly. After exploring all moves, the function returns either the maximum or minimum evaluation depending on the player's turn, effectively allowing the AI to choose the best possible move based on the evaluation of future board states.

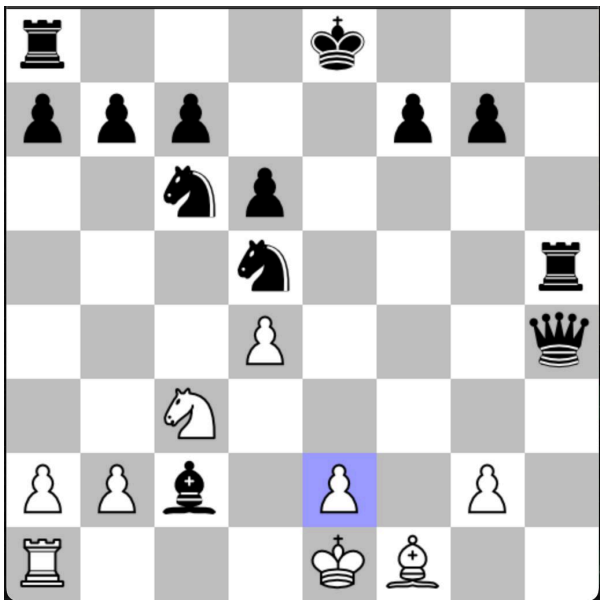
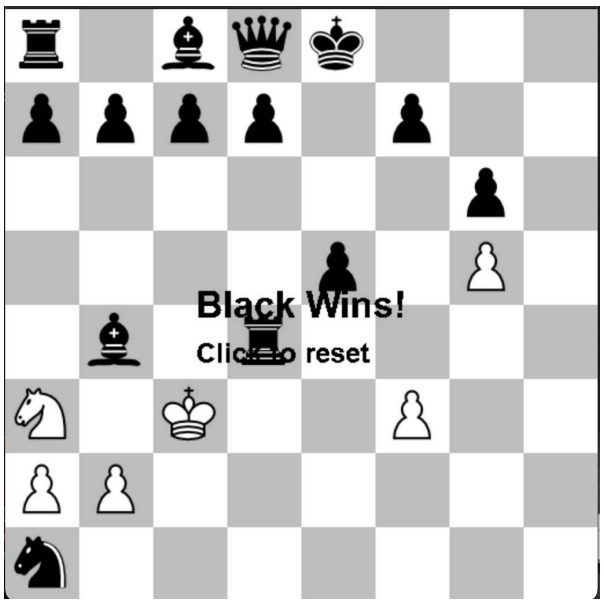
```
160  def find_best_move(gs, depth=3):
161      """Finds the best move for black using minimax with alpha-beta pruning"""
162      valid_moves = gs.getvalidmoves()
163      best_move = None
164      min_eval = float('inf')
165      alpha = float('-inf')
166      beta = float('inf')
167
168      # Randomize move order for more varied gameplay
169      valid_moves = list(valid_moves)
170      random.shuffle(valid_moves)
171
172      for move in valid_moves:
173          gs.makemove(move)
174          eval = minimax(gs, depth - 1, alpha, beta, True)
175          gs.undomove()
176
177          if eval < min_eval:
178              min_eval = eval
179              best_move = move
180          beta = min(beta, eval)
181
182      return best_move if best_move else random.choice(valid_moves)  # Fallback to random move if no good move found
```

Figure 67 : Findbestmove Function

The find_best_move function is designed to find the best move for Black using the Minimax algorithm with alpha-beta pruning. It takes two parameters: gs, the current GameState object, and depth, which defaults to 3 (indicating how many moves ahead to evaluate). The function first retrieves all valid moves for the current position using gs.getvalidmoves(). It initializes best_move as None and min_eval as positive infinity to track the best move and its associated evaluation. It also sets up the alpha (alpha = -∞) and beta (beta = ∞) values for pruning. The valid moves list is shuffled to introduce some randomness in the move order, ensuring more varied gameplay. Then, for each move, the function makes the move on the game board, evaluates the resulting position using the minimax function, and undoes the move. If the evaluation of the move is lower than min_eval, it updates min_eval and sets best_move to the current move. The beta value is updated to be the smaller of beta and the current evaluation. If no move improves upon the current best, the function returns a random move from the valid moves as a fallback, ensuring that there is always a move to play even if the evaluation mechanism doesn't find a clear best move.

Pictures (Game Rundown)

When the pieces are clicked, it highlights the possible / legal places it can move to	

Only pieces/king with the possibility of blocking the checkmate can be moved. In this case, the black queen checkmates the white king, so the white pawn in front of the king is not allowed to move because the king has to move to avoid checkmate, therefore losing the game	
When there are no more possibilities of the king avoiding being eliminated, the game stops and in this case, black (AI) wins the game	

References / Credits

The chess board building : https://www.youtube.com/@eddiesharick6649

The main idea behind chess.ai file :

- https://www.chessprogramming.org/Minimax#:~:text=an%20algorithm%20used%20to%20determine,according%20to%20an%20evaluation%20function.
- https://github.com/apostolisv/chess-ai/blob/master/Computer.py