Assignment One: Draughts

Knowledge & Reasoning

Cand.No. 183708

December 2, 2020

Contents

1 Introduction

This project entailed building a Draughts game using the object-oriented programming paradigm. This comprised of creating the game's back-end processes as well as its Graphical User Interface (GUI), to provide an interactive display with a board and also AI and player move-able pieces. This project was written in Python using Pygame, which provided the graphics libraries required to complete this task. The finished game is shown below in Figure 1.

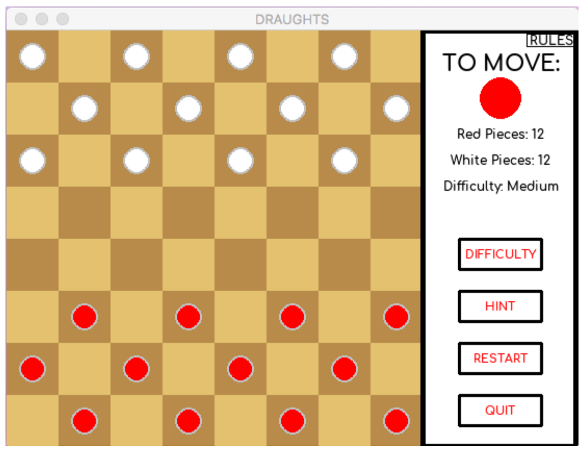

Figure 1: Display upon opening the Application

The bulk of the code is split into four main classes and also a collection of functions to handle events whilst the game is running. The four classes are Piece, GameBoard, GameManager, AI_Player. The Piece class holds all information about a particular piece such as its colour, the shape it draws on a board, and the direction that piece will move in. GameBoard stores information of a Draught's board in instance variables such as the sum of a player's pieces and their positions on a board. This class is also responsible for determining the legal moves of any particular piece and contains the heuristic evaluation functions for determining the desirability of a game state. The GameManager was created to handle functionality relating to a player's interaction with the game. For example, its methods control the selection and move process of a piece and the passing of that information down to its GameBoard instance variable and all subsequent Pieces, for drawing them on the draughts board. Contained within this class is also the functionality relating to the scorepanel (the rectangular break-away portion seen on the right of Figure 1), such as the methods which control the changing of information internally, by way of the buttons. The AI_Player class is used to create an opponent AI player and contains the help feature, where the object uses one of it's algorithms to provide hints for potential good moves, based off the current game state and future states. This class stores the opponent difficulty level the player selects to play against.

2 Program Functionality

This section details the requirements for the back-end processes within the game and describes how the implemented functionality addresses these criteria. This means certain aspects of the user interface components will be omitted, due them being easier to demonstrate by running the game itself. The game is run using the *main*() function which creates a GameManager object and an AI_Player object, and contains the main while loop in which mouse click events are handled and the AI_Player makes its moves. The game initializes with all twenty-four pieces taking positions on the black squares with the middle two rows of the 2D array left empty.

2.1 Interactivity

The scorepanel contains five buttons which effect the state of the game. The rules button which is located on the top right of the scorepanel, when pressed, will display a pop out message over the board itself which details the rules for this iteration of Draughts. To display this, the *scorepanel_rules*() method of the GameManager class is called. This pop out is shown below in Figure 2.

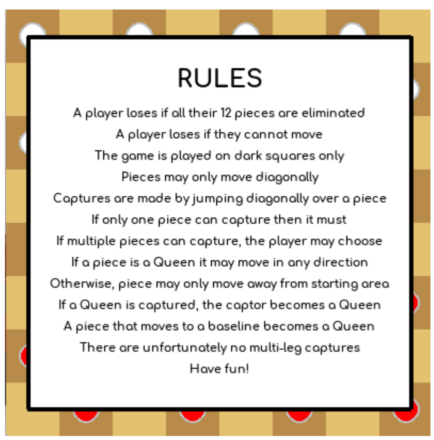

Figure 2: The rules of this specific Draughts game

The other four buttons are titled "DIFFICULTY", "HINT", "RESTART" and "QUIT" (Figure 3) and are located towards the bottom of the scorepanel. These are initialized using the *scorepanel_game_buttons*() method of the same GameManager class. All five "buttons" are merely displays and possess no interactivity by themselves; To handle events during the game such as mouse clicks on these button displays, the system makes use of the previously mentioned collection of functions. These functions are titled: *rules*(), *change_difficulty*(), *hint*(), *restart_game*() and *quit_game*(). The difficulty and hint functions have an "ai" parameter which takes the AI_Player object as its argument (which will be discussed in greater depth in a later section). The restart (which works by re-initializing part of the GameManager object, resetting select instance variables) and rules functions are passed the GameManager object. These buttons pass information back to their respective object arguments to control the state of the game. They all also have a position parameter, "pos", which makes use of the Pygame Mouse method *get_pos*() to return an x and y coordinate for the mouse's position. To call any one of these functions the mouse click has to be within a set x and y coordinate range.

Figure 3: Button Displays

For the user to move a piece around the interactive board they must mouse click on that piece's position, and then select a valid adjacent diagonal square which the piece then moves to. This is handled by the *select_piece*() and *move_piece*() methods of the GameManager class. These work in tandem by recursively selecting a piece until a selection of a piece is followed by selection of a potential square that that piece can move to. If the potential square is a valid move then the move piece method calls the GameBoard *move_piece*() method which in turn calls the Piece *move_piece*() method. If playing against a computer controlled opponent the move piece process on their side is automatic.

2.2 Validation of Moves

Validation of potential moves was achieved mainly using the GameBoard methods: *__get_valid_moves_dir*() (getMovesDir), *get_valid_moves*() (getMoves) and *get_all_pieces*(). Unfortunately this iteration of draughts is without a working multi-leg capturing system - for the rest of this report a reference to valid moves shall indicate traditional valid moves *without* double, etc. jumps. getMovesDir takes a Piece object and a direction as its parameters. The object is used to get the row and column of the piece in question. getMovesDir uses the current piece position and adds the direction argument once or twice to its row's value (± 1 or ± 2) for a next_row and next_next_row variable. The function first looks for opponent pieces occupying the adjacent diagonals using the row variables and column ± 1 or ± 2. This area under examination is shown in Figure 4.

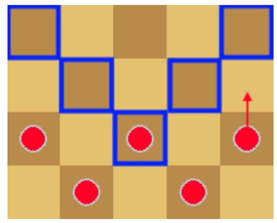

Figure 4: Annotation of area under examination in getMovesDir
Not in-game image

If an opponent is found the next diagonal along's state is also checked. If empty then the piece can capture and that position will be stored in a list. If no captures are available, a second loop will be executed to add valid non-capture moves of adjacent diagonals to the empty list. This list of valid moves is returned. getMovesDir is a private method as it should only be called in getMoves (which has only one parameter: piece). getMoves is used to account for Queen pieces which are omni-directional and so getMovesDir is applied in both directions. Once a piece is selected its valid moves are shown on the board in the form of "ghost" pieces, which are smaller, similarly coloured versions of the player's piece. As seen in Figures 5,6.

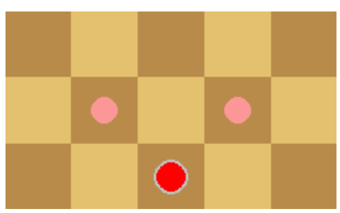

Figure 5: Valid moves with clear board

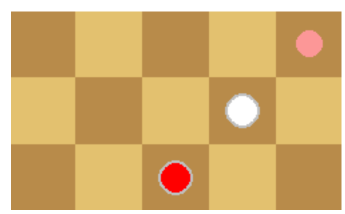

Figure 6: Valid moves with opponent

This way of showing the player what moves are available to each piece removes the need to reject invalid moves and send the player explanations as to why their move was rejected, as this can interrupt the flow of the game. However in "Forced Capture" cases a more dedicated message was deemed appropriate. If the player attempts an invalid move (due to the forced capture rules) two things happen: First the piece or pieces that are in capturing positions are highlighted with their current squares by a blue border (Figure 7). Secondly, the scorepanel will display the text "forced capture" in blue capital letters (Figure 8).

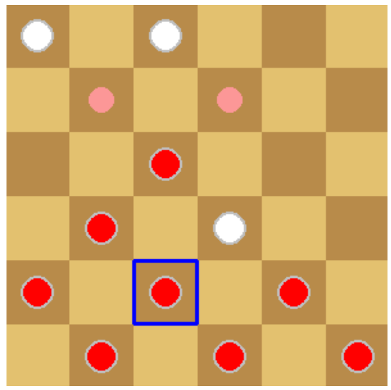

Figure 7: Blue square surrounding piece that must capture, if attempted invalid move

The forced capture system utilizes *get_all_pieces*() (getPieces). This method takes a colour parameter and an optional Boolean parameter "capture", which is defaulted to True. If capture is not true then the method will loop through all board positions, append Piece objects of the specified colour to a list and return the list of discovered pieces. If capture is true the method will iterate through the list of discovered pieces, pass each piece through the getMoves method and evaluate whether a piece is in a capturing position. If any piece is found to be in a capturing position then the list returned is no longer a list of all pieces but a list of all pieces which may capture.

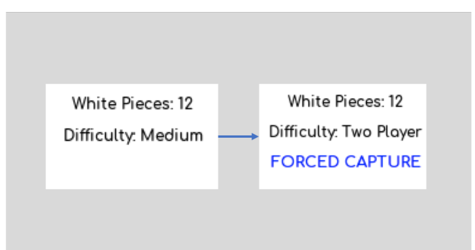

Figure 8: Annotated example of transition to scorepanel pop-up message, if attempted invalid move

The drawing of the forced capture graphics is handled in the GameManager's *move_piece*() method which calls getPieces and evaluates the player input to see if a move is invalid.

2.3 Extra Features

There are two ways for a piece to be crowned Queen: First, if a piece makes it to the opposite baseline it is crowned Queen. This event is handled in the GameBoard's *move_piece*() method which checks if a piece has moved to either board[0][x] or board[7][x]. Second, a piece that commits regicide against an opponent also becomes a Queen. This is handled in the same method by checking during a capture if the piece being captured is a Queen. Transition by regicide is shown in Figures 9 and 10.

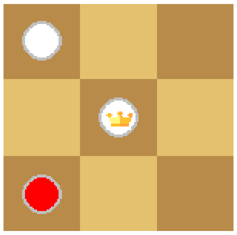

Figure 9: Moments before a heinous murder

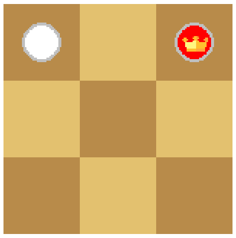

Figure 10: The Queen is dead, long live the (other) Queen!

The GameManager's scorepanel also displays information relating to the current state of the game. The turn marker component, called using *scorepanel_turn_marker*(), displays the text "TO MOVE" and a circle with the colour of the player whose move it is. Upon completion of the game (when *gameover*() is called) the turn marker will change it's text to "WINNER" and the font colour to green. The circle's colour changes to the winning colour.

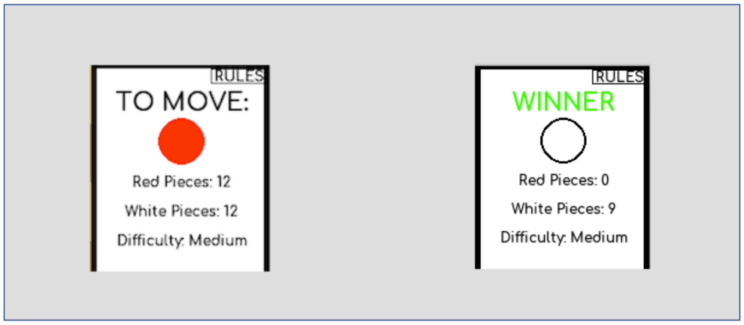

Figure 11: The top section of the game's scorepanel

Beneath the turn marker is the information relating to the board and game's state. This displays the amount of pieces remaining on both player's side and also the current difficulty level of the AI opponent. This is displayed by calling the *scorepanel_info_text*() method.

When the hint button is pressed the AI_Player method *hint_move*() is called which uses the minimax algorithm to make a prediction on a good next move for the player (Figure 12). This works for both sides in two-player mode.

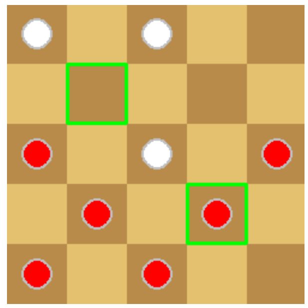

Figure 12: Hint functionality

3 Search Algorithms and AI

This game uses a 2D array stored in the GameBoard class. Indexing this array returns the pieces and empty squares used in returning a next board state. This runs in linear time: $O(\text{Rows}\times\text{Columns}) \rightarrow O(8\times8) \rightarrow O(\text{n})$. The game draughts is known to have an average branching factor of 2.8 meaning on average the player will only ≈ 2.8 legal moves per turn.

3.1 RandomAI

The AI_Player method $random_AI()$ has a colour parameter and finds all pieces on the board with that colour and also with valid moves. This method shuffles the list of pieces that meet these criteria, selects a piece, shuffles that piece's valid moves and then selects the first move option. It returns a GameBoard object augmented by the chosen move, which overwrites the GameBoard stored in the main loop's GameManager.

3.2 Minimax Algorithm with Alpha-Beta Optimization

3.2.1 The Successor Function

The *get_all_pieces_moves*() method was used to: Find all pieces on that board (of AI player colour), get valid moves for each piece, deepcopy the board and select the copied piece object from within, simulate a legal move on that board (using *imitate_move*()) and add the augmented deepcopied board to a list of potential future states to be returned. This method calls the GameBoard method *get_all_pieces*($capture = True$) so only pieces that should move (accounting for forced capture) are used.

3.2.2 Minimax

This is a recursive backtracking algorithm used for selecting the next move in an n-player game. This is a suitable algorithm for draughts due to the game's low branching factor; A game with a higher branching factor would mean more potential moves which in turn leads to an exponential increase in the number of explored nodes. This algorithm relies on players playing optimally and works by instancing a maximizer, which aims for the greater heuristic (the evaluation of the board in a given state), and a minimizer which aims to minimize this evaluation. The maximizer and minimizer simulate the progression of the game in a depth-first fashion until a node at the target depth is reached or the node is a leaf node. Each maximizer or minimizer call is a ply (a single player's move) and the search depth is increased by one. The evaluation of the target-depth or terminal nodes are passed back up the game tree and compared to the current evaluation at that depth level. If the result is greater than that value is stored. In this implementation the method returns the evaluation value and a GameBoard object with the "best move" applied to the board.

3.2.3 Alpha-Beta Pruning

This is an optimization technique for the minimax algorithm. Minimax evaluates all game tree branches and their leaf nodes with time complexity $O(b \times b \times ... \times b) = O(b^d)$ where b is the branching factor and d is the the target search depth. This pruning technique adds two extra parameters to the minimax method: α and β. These values correspond the the best value the maximizer and the minimizer can guarantee at a game-tree's depth level or above. The technique prunes off branches that need not be explored due to the known availability of a better move. In the worst case scenario, with non-optimal move ordering, $\alpha - \beta$ has the same time complexity as minimax. If move ordering is optimal and target depth level is even, this is reduced to $O(\sqrt{b^d}$. A comparison can be seen below in Figure 13.

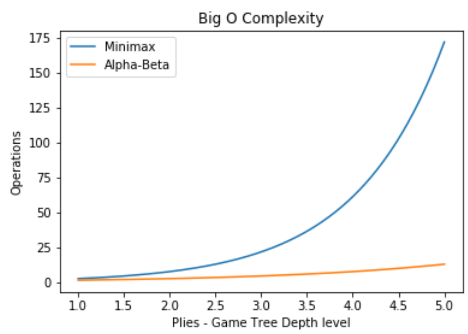

Figure 13: Comparison of operations in minimax with and without $\alpha\beta$ optimization, per ply

3.2.4 Heuristics

The heuristic function used by the minimax algorithm evaluated a board by its number of pieces and its number of queens compared to the opponents corresponding amounts. By maximising this value it aims to maximize its value in the ratio of computer to player pieces and so win the game. This was an simple but effective method. A second evaluation function was created which took into account more factors such as the positioning of pieces and assigned weights to those heuristic components. Unfortunately this never appeared to outperform the original evaluation function in user testing and so it is not the default AI function. Both $evaluate\_h1$ and $evaluate\_h2$ can be found within the GameBoard class.

3.3 Difficulty

Difficulty is changed during the game using the $change\_difficulty()$ function. When this "button" is pressed it will cycle the AI_Player's difficulty instance variable between [-1,4]. These five game modes are: "Two Player", "Easy", "Medium", "Hard". "Very Hard" and "Last Stand" and are progressively harder to beat. These were named to be familiar and intuitive to the player with the exception of "Last Stand" mode. This game mode scales the target depth of the minimax method with the amount of pieces remaining on the computer's side, with a maximum of depth eleven to imitate a final push by the AI. There are less potential states to explore when at a depth of eleven due to the diminished number of pieces still in play at the end game, making this computationally doable.

4 Conclusion

This was an informative project, learning the capabilities of the minimax algorithm and becoming familiar with the Pygame library was interesting. Some considerations for this project if I revisit it would be to correct and fine-tune the second evaluation function as the first is fairly basic. Also a method within minimax to give preference to certain game tree branches, by optimally ordering the list of moves would have been good to explore, as this would hopefully have further decreased the number of game states that would need to be explored with minimax and alpha beta.

Appendix

Code attached over leaf.
This will have been exported to .py from a Jupyter Notebook so please excuse any missed formatting issues.

```python
#!/usr/bin/env python
# coding: utf-8

# In[1]:


# !pip install pygame #(For examiner)
# pygame 2.0.0 (SDL 2.0.12, python 3.7.4)
import numpy as np
import pygame as p
import random
import copy
import time


# In[2]:


# Constants:
WIDTH, HEIGHT = 400, 400
if WIDTH <= 400:
    WIDTH = 400
if HEIGHT <= 400:
    HEIGHT = 400
ROWS, COLS, = 8, 8
BOX_SIZE = WIDTH//COLS

RED, GREEN, BLUE = (255,0,0), (0,255,0), (0,0,255)
WHITE, BLACK, SILVER = (255, 255, 255), (0,0,0), (192,192,192)
SALMON = (252,151,151)
DARK =  (184,139,74)
LIGHT = (227,193,111)

SCOREPANEL_SIZE = 150

image_dimensions = int(17/400 * WIDTH)
CROWN = p.transform.scale(p.image.load("crown.png"), (image_dimensions,image_dimensions))

RULES = ["A player loses if all their 12 pieces are eliminated",
         "A player loses if they cannot move",
         "The game is played on dark squares only",
         "Pieces may only move diagonally",
         "Captures are made by jumping diagonally over a piece",
         "If only one piece can capture then it must",
         "If multiple pieces can capture, the player may choose",
         "If a piece is a Queen it may move in any direction",
         "Otherwise, piece may only move away from starting area",
         "If a Queen is captured, the captor becomes a Queen",
         "A piece that moves to a baseline becomes a Queen",
         "There are unfortunately no multi-leg captures",
         "Have fun!"
        ]



# In[3]:


class Piece:

    PADDING = 14
    BORDER = 2

    def __init__(self, row, column, colour):
        self.row = row
        self.column = column
        self.colour = colour
        if self.colour == WHITE:
```

```
            self.direction = 1
        else:
            self.direction = -1
        self.king = False
        self.x = 0
        self.y = 0
        self.calculate_position()

    def set_king(self):
        self.king = True

    def calculate_position(self):
        self.x = BOX_SIZE * self.column + BOX_SIZE // 2
        self.y = BOX_SIZE * self.row + BOX_SIZE // 2

    def draw(self, win):
        radius = BOX_SIZE//2 - self.PADDING
        p.draw.circle(win, SILVER, (self.x,self.y), radius + self.BORDER)
        p.draw.circle(win, self.colour, (self.x,self.y), radius)
        if self.king:
            new_x = self.x - CROWN.get_width()//2
            new_y = self.y - CROWN.get_height()//2
            win.blit(CROWN, (new_x, new_y))

    def move_piece(self, row, column):
        self.row = row
        self.column = column
        self.calculate_position()

    def __repr__(self):
        if self.direction > 0:
            return "+"
        else:
            return "-"


# In[4]:


class GameBoard:


    def __init__(self):
        self.board = np.zeros(shape=(8,8)).astype("int").tolist()
        self.red_remaining = self.white_remaining = 12
        self.red_king_count = self.white_king_count = 0
        self.setup_board()


    def setup_board(self):
        """
        Fills the 2d array with piece objects corresponding to draughts positions.
        """
        # Start player 2 at "top" of board.
        colour = 2
        # Counter to switch to using player 1 identifier.
        counter = 0
        ln = len(self.board)
        for row_index in range(ln):
          if self.is_even(row_index) and row_index != 4:
            self.board[row_index] = [colour,0,colour,0,colour,0,colour,0]
          elif not self.is_even(row_index) and row_index != 3:
            self.board[row_index] = [0,colour,0,colour,0,colour,0,colour]
          counter += 1
          if counter == 4:
            colour = 1
        # Messy Setup.
        for row_index in range(ln):
```

```
          for col_index in range(ln):
            if self.board[row_index][col_index] != 0:
              colour = self.board[row_index][col_index]
              if colour == 1:
                colour = RED
              else:
                colour = WHITE
              self.board[row_index][col_index] = Piece(row_index, col_index, colour)


    def draw_board(self, win):
        win.fill(LIGHT)
        for row in range(ROWS):
          for col in range(row % 2, COLS, 2):
            p.draw.rect(win, DARK, (row*BOX_SIZE, col*BOX_SIZE, BOX_SIZE, BOX_SIZE))


    def draw_all(self, win):
        # Draw board.
        self.draw_board(win)
        # Draw pieces on board.
        for row in range(ROWS):
          for col in range(COLS):
            piece = self.board[row][col]
            if piece != 0:
              piece.draw(win)


    def get_valid_moves(self, piece):
        """
        Get all valid moves for a particular piece.
        Param: piece (Piece): The piece to find valid moves for.
        Return: full_moves (List): The list of valid moves for that piece.
        """
        full_moves = []
        if piece.king:
            for direction in [-1,+1]:
                full_moves.append(self.__get_valid_moves_dir(piece,direction))
        else:
            full_moves.append(self.__get_valid_moves_dir(piece,piece.direction))
        full_moves = self.flatten(full_moves)
        for move in full_moves:
            if self.can_capture(piece.row, move[0]):
                full_moves = [move for move in full_moves if self.can_capture(piece.row, move[0])]
                break

        return full_moves

    def __get_valid_moves_dir(self, piece, direction):
        """
        Get all valid moves for a particular piece in a particular direction.
        Param: piece (Piece): The piece to find valid moves for.
        Param: direction (int): The direction to traverse the 2d array: +/- 1.
        Return: next_move_list (List): List of valid moves for a piece in a certain direction.
        """
        # End function if non player piece.
        try:
          # Get row, column (indicies) from tuple object piece.
          row, column = piece.row, piece.column
        except AttributeError:
            print("No valid moves for an empty space!")
            return
        # Potential next moves list.
        next_move_list = []
        # Next row - dependent on player.
        next_row = row + direction
```

```
        next_next_row = row + direction + direction
        # List to hold columns on either side.
        left_right = [column-1,column+1]
        # Loop through left right options.
        for next_col in left_right:
          if next_col in range(8) and next_row in range(8):
            next_space = self.whats_in_the_box(next_row, next_col)
            if isinstance(next_space, Piece):
              # Split conditional - case next_space=0, no int attribute colour.
              if next_space.colour != piece.colour:
              # Assign next next column indicies.
                next_next_col = None
                if next_col == column - 1:
                    next_next_col = column - 2
                else:
                    next_next_col = column + 2
                if next_next_col in range(8) and next_next_row in range(8):
                    if self.whats_in_the_box(next_next_row,next_next_col) == 0:
                        next_move_list.append((next_next_row, next_next_col))
        # If no forced capture moves yet.
        if not next_move_list:
          # Case: Empty square.
          for next_col in left_right:
            if next_col in range(8) and next_row in range(8):
              # Check state of potential next square.
              if self.whats_in_the_box(next_row, next_col) == 0:
                next_move_list.append((next_row, next_col))
        return next_move_list


    def can_capture(self, row, new_row):
        if new_row in [row+2, row-2]:
            return True
        return False


    def move_piece(self, Piece, new_row, new_col):
        """
        Move a piece to a new position on the board - also handles removing captured pieces.
        Param Piece (Piece): The piece to move.
        Param new_row, new_col (int): New row/column to move piece to.
        Return Boolean: For use in GameManager method move_piece().
        """
        # Temp. remove (parameter) Piece.
        row, col = Piece.row, Piece.column
        self.remove_piece(row,col)
        Piece.move_piece(new_row, new_col)
        self.board[new_row][new_col] = Piece
        # King update.
        if (new_row == 0 or new_row == ROWS - 1) and Piece.king != True:
            Piece.set_king()
            if Piece.colour == RED:
                self.red_king_count += 1
            else:
                self.white_king_count += 1
        # Remove opponent piece:
        if new_row in [row+2, row-2]:
            x = (row+new_row)//2
            y = (col+new_col)//2
            opp_piece = self.whats_in_the_box(x,y)
            if opp_piece.colour == RED:
                self.red_remaining -= 1
                if opp_piece.king:
                    Piece.set_king()
                    self.white_king_count += 1
                    self.red_king_count -= 1
            else:
                self.white_remaining -=1
```

```python
                if opp_piece.king:
                    Piece.set_king()
                    self.red_king_count += 1
                    self.white_king_count -= 1
            self.remove_piece(x,y)
              # Return True if piece is taken.
            return True
        # Else False
        return False


    def whats_in_the_box(self, row, column):
        return self.board[row][column]

    def remove_piece(self, row, col):
        self.board[row][col] = 0

    def get_all_pieces(self, colour, capture=True):
        """
        Get all Piece objects for a given colour or get all piece objects that can capture.
        Param colour (Tuple): (x,x,x) RGB value of piece's colour.
        Param capture (Boolean): Defaulted to true - for returning only pieces that can capture.
        Return pieces_can_capture (List): Returns only pieces in positions that can capture.
        Return total (List): Return all piece objects of a given colour.
        """
        total = []
        for row in self.board:
            for item in row:
                if isinstance(item, Piece) and item.colour == colour:
                    total.append(item)
        if capture:
            pieces_can_capture = []
            for piece in total:
                for move in self.get_valid_moves(piece):
                    if move:
                        if self.can_capture(piece.row, move[0]):
                            pieces_can_capture.append(piece)
            if pieces_can_capture:
                return pieces_can_capture
        return total


    def can_colour_move(self, colour):
        pieces = self.get_all_pieces(colour, capture=False)
        can_move = False
        for piece in pieces:
            if self.get_valid_moves(piece):
                can_move = True
                break
        return can_move

    def gameover(self):
        """
        Return the winning player if either side can no longer move or has 0 pieces left.
        """
        if self.red_remaining == 0 or not self.can_colour_move(RED):
            return WHITE
        elif self.white_remaining == 0 or not self.can_colour_move(WHITE):
            return RED
        else:
            return False

    def is_even(self,num):
        return (num % 2) == 0

    def flatten(self,ls):
        return [item for m_ls in ls for item in m_ls]
```

```python

    def print_board(self):
        print()
        for row in self.board:
            print(row)
        print()


    def evaluate(self, colour, method="h1"):
        """
        Select heuristic.
        """
        if method == "h1":
            return self.evaluate_h1(colour)
        elif method == "h2":
            return self.evaluate_h2(colour)

    def evaluate_h1(self, colour):
        """
        Heuristic evaluation for AI player to determine desirability of future board states.
        Param colour (Tuple): RGB - Evaluate goodness of board for a specific colour.
        Return (int): Integer evaluation of a board for use in minimax method.
        """
        if colour == WHITE:
            return self.white_remaining - self.red_remaining + (self.white_king_count - self.red_king_count)
        else:
            return self.red_remaining - self.white_remaining + (self.red_king_count - self.white_king_count)


    # Unfortunately I was unable to tweak h2 to a sufficient level where I felt
    # it outperformed h1. Therefore h1 is the default heuristic function.
    def evaluate_h2(self, colour):
        """
        Heuristic evaluation for AI player to determine desirability of future board states.
        Param colour (Tuple): RGB - Evaluate goodness of board for a specific colour.
        Return (int): Integer evaluation of a board for use in minimax method.
        """
        weight = {"backrow":0.1,
                  "safe_edges":0.1,
                  "queens":0.2,
                  "pieces":0.5,
                  "avoid_forfeit":0.1
                  }
        evaluation = 0
        backrow_value = self.heuristic_backrow(colour) * weight["backrow"]
        safe_edges_value = self.heuristic_safe_edges(colour) * weight["safe_edges"]
        queens_value = self.heuristic_queens(colour) * weight["queens"]
        pieces_value = self.heuristic_pieces(colour) * weight["pieces"]
        avoid_forfeit_value = self.heuristic_avoid_forfeit(colour) * weight["avoid_forfeit"]
        evaluation = sum([backrow_value, safe_edges_value, queens_value, pieces_value, avoid_forfeit_value])
        return evaluation


    def heuristic_backrow(self, colour):
        """
        Value for pieces on backrow.
        """
        value = 0
        if colour == WHITE:
            for piece in self.board[0]:
                if piece:
                    if piece.colour == WHITE:
                        value += 1
        elif colour == RED:
            for piece in self.board[ROWS-1]:
```

```python
                if piece:
                    if piece.colour == RED:
                        value += 1
        return value

    def heuristic_safe_edges(self, colour):
        """
        Value for pieces safe on edges.
        """
        value = 0
        for i in range(8):
            piece_left = self.board[i][0]
            if piece_left:
                if piece_left.colour == colour:
                    value += 1
            piece_right = self.board[i][7]
            if piece_right:
                if piece_right.colour == colour:
                    value += 1
        return value

    def heuristic_queens(self, colour):
        """
        Value for number of queen pieces.
        """
        value = 0
        if colour == WHITE:
            value = self.white_king_count * 0.75 - self.red_king_count * 0.75
        if colour == RED:
            value = self.red_king_count * 0.75 - self.white_king_count * 0.75
        return value

    def heuristic_pieces(self, colour):
        """
        Value for number of pieces.
        """
        value = 0
        if colour == WHITE:
            value = self.white_remaining - self.red_remaining
        if colour == RED:
            value = self.red_remaining - self.white_remaining
        return value

    def heuristic_avoid_forfeit(self, colour):
        """
        Value for avoiding a cannot move position on board.
        """
        value = 5
        if self.can_colour_move(colour):
            return value

    def heuristic_control_centre_board(self, colour):
        """
        Value for controlling centre of board.
        """
        NotImplemented

    def heuristic_in_capturable_position(self, colour):
        """
        Value for piece in position to get captured.
        """
        NotImplemented

    def heuristic_in_capturing_position(self, colour):
        """
        Value for piece in position to capture.
        """
        NotImplemented
```

```


# In[5]:


class GameManager:

    def __init__(self, win, scorepanel_size, difficulty=1, turn=RED):
        self.win = win
        self.scorepanel_size = scorepanel_size
        self.difficulty = difficulty
        self.__init()
        self.turn = turn

    def __init(self):
        """
        Partial reinitialization for reseting the game state - start new game.
        Called in reset_game().
        """
        self.selected_piece = None
        self.gameboard = GameBoard()
        self.valid_moves = []
        self.turn = RED
        self.hint_squares = []
        self.show_hint = False
        self.can_capture_pieces = []
        self.correct_moves_assist = False
        self.rules_button_pressed = False

    def reset_game(self):
        self.__init()

    def update(self):
        """
        Draw any board/game changes with pygame.
        """
        self.gameboard.draw_all(self.win)
        self.draw_valid_moves(self.valid_moves)
        self.scorepanel()
        if self.rules_button_pressed:
            self.display_rules()
        self.draw_hints()
        if self.correct_moves_assist:
            self.draw_correct_moves()
        p.display.update()
        p.display.flip()


    def select_piece(self, row, col):
        """
        Recursive method to handle user interaction with pieces in GUI - Method runs
    continously in main game loop.
        Param row, col (int): Position indicies for selecting and moving a piece.
        Return (Boolean): If selection valid return true, otherwise false.
        """
        # Allow for continous selection of pieces.
        if self.selected_piece:
            self.show_hint = False
            move = self.move_piece(row, col)
            if not move:
                # Reset  selection.
                self.selected_piece = None
                self.select_piece(row,col)
        piece = self.gameboard.whats_in_the_box(row, col)
        if piece.colour == self.turn:
            self.selected_piece = piece
            self.valid_moves = self.gameboard.get_valid_moves(piece)
            return True
```

```
        return False


    def move_piece(self, row, col):
        """
        Method to handle moving a piece, called in select_piece(). Also handles forced capture system.
        Param: row, col (int): Position indicies for attempting to move a piece to new position.
        Return (Boolean): If piece does not move return false otherwise true. Used in select_piece().
        """
        # N.B. In instances when no pieces can capture then can_capture_pieces will contain all pieces
        self.can_capture_pieces = self.gameboard.get_all_pieces(self.turn,capture=True)
        piece = self.gameboard.whats_in_the_box(row, col)
        # If selected piece, piece (potential move) is not a piece, and (row,col) is valid move then move.
        if self.selected_piece and not isinstance(piece, Piece) and (row, col) in self.valid_moves:
            # If selected piece is in capture_pieces.
            if self.selected_piece in self.can_capture_pieces:
                self.gameboard.move_piece(self.selected_piece, row, col)
                # If move is successful switch current turn.
                self.turn_switch()
            # If selected piece is not in capturing moves (will only apply if there are capturing moves as otherwise all pieces are in capture_moves)
            else:
                # Bool switched on to display forced capture.
                self.correct_moves_assist = True
                return False
        else:
            return False
        return True


    def turn_switch(self):
        """
        Switch control to other player. Also resets instance variables for the next player to make use of.
        """
        self.can_capture_pieces = []
        self.correct_moves_assist = False
        self.hint_squares = []
        self.valid_moves = []
        if self.turn == RED:
            self.turn = WHITE
        else:
            self.turn = RED

    def turn_ai(self, board):
        """
        Gets the GameBoard object from the AI_Player's prediction and overwrites current Gameboard (updates for AI move).
        Param board (GameBoard): Overwrite class's instance variable with new board.
        """
        self.gameboard = board
        self.turn_switch()

    def get_board(self):
        return self.gameboard


    def draw_valid_moves(self, valid_moves):
        for pos_move in valid_moves:
            row, col = pos_move
            circle_x = col*BOX_SIZE + BOX_SIZE//2
            circle_y = row*BOX_SIZE + BOX_SIZE//2
```

```python
            colour = None
            if self.turn == RED:
                colour = SALMON
            else:
                colour = SILVER
            p.draw.circle(self.win, colour, (circle_x, circle_y), 10)

    def draw_hints(self):
        """
        Draw green hint sqaures around the piece to move and the next move.
        """
        if self.show_hint:
            row, col = self.hint_squares[0][0], self.hint_squares[0][1]
            p.draw.rect(self.win, GREEN, (col*BOX_SIZE, row*BOX_SIZE, BOX_SIZE, BOX_SIZE), width=3)
            # Suggested move.
            row, col = self.hint_squares[1][0], self.hint_squares[1][1]
            p.draw.rect(self.win, GREEN, (col*BOX_SIZE, row*BOX_SIZE, BOX_SIZE, BOX_SIZE), width=3)

    def draw_correct_moves(self):
        """
        Draw blue forced capture square around the piece that must capture.
        """
        for piece in self.can_capture_pieces:
            p.draw.rect(self.win, BLUE, (piece.column*BOX_SIZE, piece.row*BOX_SIZE, BOX_SIZE, BOX_SIZE), width=3)

    def scorepanel(self):
        """
        Draw the scorepanel shape and all of its components.
        """
        # Panel shape.
        p.draw.rect(self.win, WHITE, (WIDTH,0,self.scorepanel_size-2,HEIGHT))
        p.draw.rect(self.win, BLACK, (WIDTH,0,self.scorepanel_size-2,HEIGHT), self.scorepanel_size//30)
        # Turn marker.
        self.scorepanel_turn_marker()
        # Rules button.
        self.scorepanel_rules_button()
        # "DIFFICULTY", "HINT", "RESTART", "QUIT" buttons.
        self.scorepanel_game_buttons()
        # Invalid Move message.
        self.scorepanel_forced_capture_popup()
        # Pieces Remaining + Difficulty level.
        self.scorepanel_info_text()
        # Debugging.
        # self.scorepanel_button(str(self.valid_moves), HEIGHT - <Y>)


    def scorepanel_button(self, button_text, y_pos, text_size=16, text_colour=RED, rect_colour=BLACK, border=3, centre=None, rect_height=30):
        """
        Avoid redunency with a method to draw most scorepanel components.
        Mostly handles superimposing text on a rectangular button.
        """
        smallfont = p.font.Font('Comfortaa-Regular.ttf',text_size)
        text = smallfont.render(button_text , True , text_colour)
        if centre == None:
            centre=WIDTH+self.scorepanel_size/2
        displacement = 40
        p.draw.rect(self.win, rect_colour, (centre - displacement, y_pos,displacement*2, rect_height), border)
        text_rect = text.get_rect(center=(centre, y_pos+(rect_height//2)))
        self.win.blit(text, text_rect)

    def scorepanel_turn_marker(self):
        txt = None
```

```
        circle_colour = None
        txt_colour = BLACK
        if not self.gameboard.gameover():
            txt = "TO MOVE:"
            circle_colour = self.turn
        else:
            txt = "WINNER"
            txt_colour = GREEN
            circle_colour = self.gameboard.gameover()
        circle_x, circle_y = WIDTH + (self.scorepanel_size/2), 65
        self.scorepanel_button(txt, circle_y-48, 21, txt_colour, WHITE, 0)
        if circle_colour == RED:
            p.draw.circle(self.win, RED, (circle_x, circle_y), 20)
        else:
            p.draw.circle(self.win, BLACK, (circle_x, circle_y), 20, 2)

    def scorepanel_info_text(self):
        red_rem = "Red Pieces: {}".format(self.gameboard.red_remaining)
        white_rem = "White Pieces: {}".format(self.gameboard.white_remaining)
        difficulty_dict = {-1:"Two Player",
                            0:"Easy",
                            1:"Medium",
                            2:"Hard",
                            3:"Very Hard",
                            4:"Last Stand"}
        difficulty_text = "Difficulty: {}".format(difficulty_dict[self.difficulty])
        y = 88
        for text in [red_rem, white_rem, difficulty_text]:
            self.scorepanel_button(text, y, 12, BLACK, WHITE, rect_height=25)
            y+=25

    def scorepanel_game_buttons(self):
        # "DIFFICULTY", "HINT", "RESTART", "QUIT" buttons.
        y = -200
        for text in ["DIFFICULTY", "HINT", "RESTART", "QUIT"]:
            self.scorepanel_button(text, HEIGHT + y, text_size=11)
            y += 50

    def scorepanel_forced_capture_popup(self):
        # Invalid Move message - Must force capture.
        if self.correct_moves_assist:
            self.scorepanel_button("FORCED CAPTURE", HEIGHT-238, text_size=13, text_colour=
    BLUE, rect_colour=WHITE, rect_height=25)

    def scorepanel_rules_button(self):
        smallfont = p.font.Font('Comfortaa-Regular.ttf',12)
        text = smallfont.render("RULES" , True , BLACK)
        p.draw.rect(self.win, BLACK, (WIDTH+self.scorepanel_size-50, 3, 48,13), 1)
        text_rect = text.get_rect(center=(WIDTH+self.scorepanel_size-26,10))
        self.win.blit(text, text_rect)

    def display_rules(self):
        centre = WIDTH//2
        displacement = 180
        # Border and page.
        p.draw.rect(self.win, WHITE, (centre-displacement, 25, displacement*2, 350), 0)
        p.draw.rect(self.win, BLACK, (centre-displacement, 25, displacement*2, 350), 4)
        # Title
        self.scorepanel_button("RULES", 25 + 25, 23, BLACK, WHITE, 0, centre)
        # Rules
        y_initial = 50 + 15
        step = 20
        for rule in RULES:
            self.scorepanel_button(rule, y_initial+step, 11, BLACK, WHITE, 0, centre,
    rect_height=25)
            step += 20

    def set_difficulty(self, difficulty):
```

```python
        self.difficulty = difficulty

    def set_hint_squares(self, hint_list):
        if self.show_hint:
            self.show_hint = False
        else:
            self.show_hint = True
        self.hint_squares = hint_list

    def set_rules_button(self):
        if self.rules_button_pressed:
            self.rules_button_pressed = False
        else:
            self.show_hint = False
            self.rules_button_pressed = True


# In[6]:


class AI_Player:

    def __init__(self, gamemanager, difficulty=1):
        self.difficulty = difficulty
        self.gamemanager = gamemanager
        self.recursive_calls = 0

    def update_gamemanager(self, gamemanager):
        self.gamemanager = gamemanager

    def set_difficulty(self, difficulty):
        self.difficulty = difficulty

    def reset_recursive_calls(self):
        self.recursive_calls = 0

    def hint_move(self, depth_level):
        """
        Compare current board with a predicted board using minimax.
        The difference in the two identifies the move the user should take based on the hint
    .
        Param: depth_level (int): The target depth level for the minimax algo to explore.
        Return (original_position, new_position) (Tuple): Row and column indicies of old and
     new move.
        """
        player1 = None
        player2 = None
        if self.gamemanager.turn == RED:
            player1 = RED
            player2 = WHITE
        else:
            player1 = WHITE
            player2 = RED
        original_board = self.gamemanager.get_board()
        _, new_board = self.minimax(original_board, depth_level, True, self.gamemanager,
                                   maxi_colour=player1, mini_colour=player2)
        original_pieces = original_board.get_all_pieces(player1, capture=False)
        new_pieces = new_board.get_all_pieces(player1, capture=False)
        original_positions_list = []
        new_positions_list = []
        for _, (original_piece,new_piece) in enumerate(zip(original_pieces,new_pieces)):
            original_positions_list.append((original_piece.row, original_piece.column))
            new_positions_list.append((new_piece.row,new_piece.column))
        original_position = None
        new_position = None
        for position in original_positions_list:
            if position not in new_positions_list:
                original_position = position
```

```python
                break
        for position in new_positions_list:
            if position not in original_positions_list:
                new_position = position
                break
        return (original_position, new_position)


    def ai_move(self, print_calls=False):
        """
        Have computer opponent select a move. Will use different methods depending on difficulty level.
        Param print_calls (Boolean): Controls whether no. of recursive calls per move is printed.
        Return new_board (GameBoard): The new gameboard object to overwrite the current.
        """
        # Lets give the appearance of slow human decision making.
        time.sleep(0.5)
        new_board = None
        evaluation = None
        # Diff = -1 -> PvP
        # Diff = 0 -> Easy
        if self.difficulty == 0:
            new_board  = self.random_AI()
        # Diff = 1 -> Medium
        elif self.difficulty == 1:
            evaluation, new_board = self.minimax(self.gamemanager.get_board(), 2, True, self.gamemanager)
        # Diff = 2 -> Hard
        elif self.difficulty == 2:
            evaluation, new_board = self.minimax(self.gamemanager.get_board(), 5, True, self.gamemanager)
        # Diff = 3 -> Very Hard
        elif self.difficulty == 3:
            evaluation, new_board = self.minimax(self.gamemanager.get_board(), 7, True, self.gamemanager)
        # Diff = 4 -> Last Stand - Progressively harder
        elif self.difficulty == 4:
            initial = 7
            add = 0
            opponent_remaining = len(self.gamemanager.gameboard.get_all_pieces(WHITE,capture=False))
            if opponent_remaining <= 3:
                add = 5
            elif opponent_remaining <= 5:
                add = 3
            elif opponent_remaining <= 8:
                add = 1
            evaluation, new_board = self.minimax(self.gamemanager.get_board(), initial+add, True, self.gamemanager)
        # print(evaluation)
        if print_calls:
            print(self.recursive_calls)
        self.reset_recursive_calls()
        return new_board


    def random_AI(self, colour=WHITE):
        """
        random_AI has computer opponent make a random move choice from all pieces' valid moves.
        Param colour (Tuple): RGB value to determine which side it is making move for.
        Return new_board (GameBoard): The new gameboard object to overwrite the current.
        """
        AI_pieces = self.gamemanager.gameboard.get_all_pieces(colour)
        AI_pieces = [x for x in AI_pieces if self.gamemanager.gameboard.get_valid_moves(x)]
        ls = list(range(len(AI_pieces)))
        random.shuffle(ls)
```

```python
        r_num = ls[0]
        random_piece = AI_pieces[r_num]
        new_board = copy.deepcopy(self.gamemanager.gameboard)
        random_piece = new_board.board[random_piece.row][random_piece.column]
        moves = new_board.get_valid_moves(random_piece)
        random.shuffle(moves)
        new_row, new_col = moves[0][0], moves[0][1]
        new_board.move_piece(random_piece, new_row, new_col)
        return new_board



    def minimax(self, board, depth, maximiser, gamemanager,
                alpha=float("-inf"), beta=float("inf"),
                maxi_colour=WHITE, mini_colour=RED):
        """
        Minimax Algorithm with alpha-beta pruning optimization.
        Param: board (GameBoard): board to use as starting point for exploring game tree.
        Param: depth (int): Target depth level to evaluate (leaf) nodes at.
        Param: maximiser (Boolean): Initially passed a colour (Tuple), following first call
    keeps track of minimizer and maximizer.
        Return evaluation (int): Evalutation using the board's heuristic.
        Return best_move (GameBoard): Return GameBoard with best move on board.
        """
        self.recursive_calls += 1
        if depth == 0 or board.gameover():
            return board.evaluate(maxi_colour), board
        if maximiser:
            max_evaluation = float("-inf")
            best_move = None
            for move in self.get_all_pieces_moves(board, maxi_colour):
                # [0] to return only board evaluation value.
                evaluation = self.minimax(move, depth-1, False, gamemanager, alpha, beta,
                                          maxi_colour, mini_colour)[0]
                max_evaluation = max(max_evaluation, evaluation)
                if max_evaluation == evaluation:
                    best_move = move
                alpha = max(alpha, max_evaluation)
                if beta <= alpha:
                    break
            return max_evaluation, best_move
        else:
            min_evaluation = float("inf")
            best_move = None
            for move in self.get_all_pieces_moves(board, mini_colour):
                # [0] to return only board evaluation value.
                evaluation = self.minimax(move, depth-1, True, gamemanager, alpha, beta,
                                         maxi_colour, mini_colour)[0]
                min_evaluation = min(min_evaluation, evaluation)
                if min_evaluation == evaluation:
                    best_move = move
                beta = min(beta, min_evaluation)
                if beta <= alpha:
                    break
            return min_evaluation, best_move


    def imitate_move(self, piece, move, board):
        """
        Imitate a move on a deepcopied board.
        Param piece (Piece): Piece object which makes the move.
        Param move (Tuple): Position indices (row/column) of new move.
        Param board (GameBoard): GameBoard's board to move the piece on.
        Return board (Gameboard): Return board with new move.
        """
        new_row, new_col = move[0], move[1]
        board.move_piece(piece, new_row, new_col)
        return board
```

```

    def get_all_pieces_moves(self, board, colour):
        """
        Get all future board states after trying valid moves from all pieces.
        Param board (GameBoard): Current working board to get piece objects and valid moves
    from.
        Param colour (Tuple): RGB value for the side calling minimax.
        Return potential_boards_list (List(GameBoard)): List of all valid future board
    states from a given board.
        """
        potential_boards_list = []
        potential_pieces = board.get_all_pieces(colour)
        for piece in potential_pieces:
            valid_moves = board.get_valid_moves(piece)
            for move in valid_moves:
                temp_board = copy.deepcopy(board)
                temp_piece = temp_board.whats_in_the_box(piece.row, piece.column)
                new_board = self.imitate_move(temp_piece, move, temp_board)
                potential_boards_list.append(new_board)
        return potential_boards_list


# In[7]:


def get_mouse_pos(pos):
    x, y = pos
    row = y // BOX_SIZE
    col = x // BOX_SIZE
    return row, col

def restart_game(pos, gamemanager):
    centre = WIDTH + (scorepanel_size/2)
    if centre - 30 <= pos[0] <= centre + 30 and HEIGHT - 100 <= pos[1] <= HEIGHT - 70:
        gamemanager.reset_game()

def quit_game(pos):
    centre = WIDTH + (scorepanel_size/2)
    if centre - 30 <= pos[0] <= centre + 30 and HEIGHT - 50 <= pos[1] <= HEIGHT - 20:
        return False
    else:
        return True

def change_difficulty(pos, ai):
    centre = WIDTH + (scorepanel_size/2)
    if centre - 30 <= pos[0] <= centre + 30 and HEIGHT - 200 <= pos[1] <= HEIGHT - 170:
        new_diff = ai.difficulty + 1
        if new_diff not in range(-1,5):
            new_diff = -1
        ai.set_difficulty(new_diff)
        ai.gamemanager.set_difficulty(new_diff)

def hint(pos, ai, depth_level=3):
    centre = WIDTH + (scorepanel_size/2)
    if centre - 30 <= pos[0] <= centre + 30 and HEIGHT - 150 <= pos[1] <= HEIGHT - 120 and
    not ai.gamemanager.gameboard.gameover():
        ai.gamemanager.set_hint_squares(ai.hint_move(depth_level))

def rules(pos,gamemanager):
    centre = WIDTH + scorepanel_size - 25
    if centre - 25 <= pos[0] <= centre + 25 and 0 <= pos[1] <=  16 and not gamemanager.
    rules_button_pressed:
        gamemanager.set_rules_button()
    elif gamemanager.rules_button_pressed:
        if not (WIDTH//2 - 180 <= pos[0] <= WIDTH//2 + 180) or not (25 <= pos[1] <= 350):
            gamemanager.set_rules_button()


```

```
# In[8]:


scorepanel_size = SCOREPANEL_SIZE
if scorepanel_size <= 110:
    scorepanel_size = 110
elif scorepanel_size > 200:
    scorepanel_size = 200

p.display.init()
p.font.init()
SCREENSIZE = (WIDTH+scorepanel_size, HEIGHT)
WIN = p.display.set_mode(SCREENSIZE)
p.display.set_caption("DRAUGHTS")
p.mouse.set_cursor(*p.cursors.tri_left)
FPS = 30

hint_depth_lvl = 4
# RED or WHITE
starting_player = RED


def main():
    run = True
    clock = p.time.Clock()
    gm = GameManager(WIN, scorepanel_size, turn=starting_player)
    opponent = AI_Player(gm)
    AI_player = True


    while run:
        # Maintain constant frames/second.
        clock.tick(FPS)

        if gm.turn == WHITE and AI_player and not gm.gameboard.gameover() and opponent.
    difficulty != -1:
            opponent.update_gamemanager(gm)
            new_board = opponent.ai_move(print_calls=False) # True to print recursive calls.
            gm.turn_ai(new_board)

        # Look for events during run.
        for event in p.event.get():
            # Non button quit:
            if event.type == p.QUIT:
                run = False
            # Mouse click events:
            if event.type == p.MOUSEBUTTONDOWN:
                pos = p.mouse.get_pos()
                row, col = get_mouse_pos(pos)
                try:
                    gm.select_piece(row,col)
                except:
                    pass
                # Restart Game.
                restart_game(pos, gm)
                # Quit Game.
                run = quit_game(pos)
                # Change difficulty.
                change_difficulty(pos, opponent)
                # Hint.
                hint(pos, opponent, depth_level=hint_depth_lvl)
                # Rules.
                rules(pos, gm)


        gm.update()

    p.display.quit()
```

```
main()
```