UNIVERSITY OF SUSSEX

Knowledge & Reasoning

Programming Project - Assignment 1 Checkers game

University of Sussex

Candidate Number : 221455

1st Dec 2021

Contents

1. Introduction

This program is an 8x8 checkers game implemented with an AI enemy player. The AI enemy player can choose the best move through Minimax algorithm optimization with Alpha-beta pruning. This program's AI can search 3(easy), 5(normal), and 7(hard) moves ahead depends on game mode setting. The player is able to confirm the current depth value on the caption.
The game is built with python&pygame cross-platform for a fully interactive GUI as an object-oriented way. Each player starts the game with 12 tokens same as the initial state of the normal checkers game. At the start of the game, the human player can move first with the RED-colored token. Each player takes turns to make a move with the proper successor function. All token's move can be handled by the user through mouse as click to select&click to place. It is easy to notice the human player's turn comes because the AI player makes a sound every time he moves the token. If a player tries to take a move that violates the successor function, the program will not follow&update that move. The player can adjust the game difficulty just press the buttons above, and can confirm the checkers rules. The game ends when the winner comes out. A player can be the winner when the player captures all of the other player's tokens.
All of these programs consist of a total of 11 .py files&assets and can be executed by simply running main.py after installing the pygame library.
Below is the snapshot of the checkers game(Figure 1):

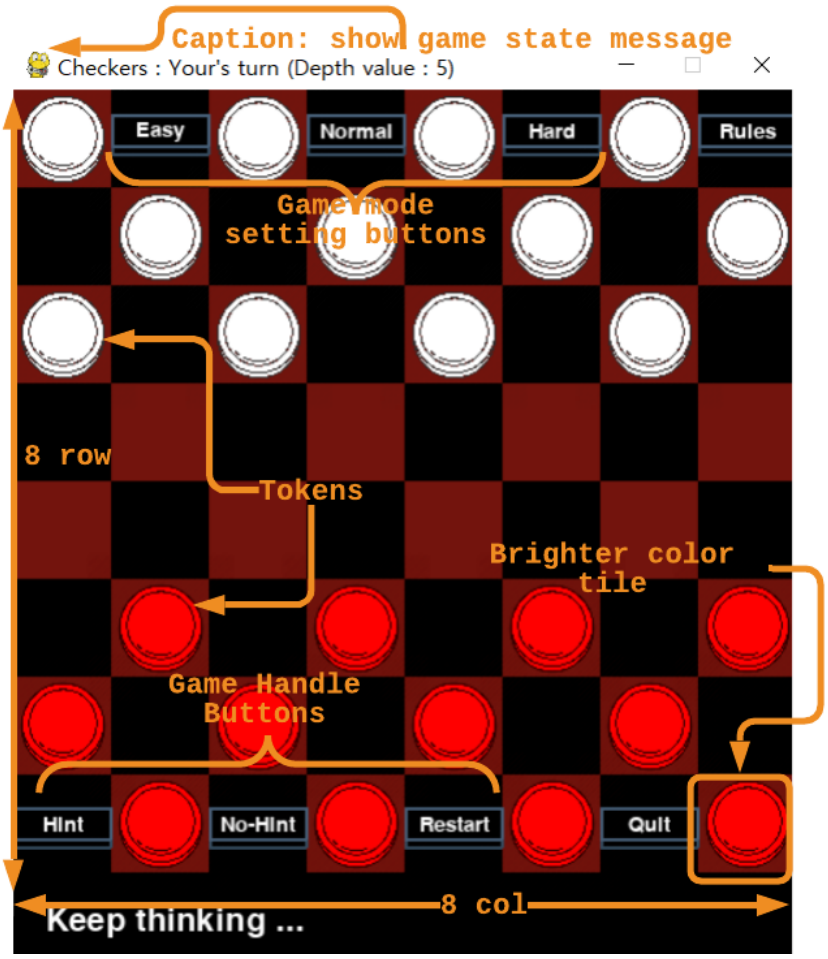

Figure 1: Snapshot of checkers game

This checkers program consists of 11 py. files

- **Main.py** [1]: The main.py is the entry point of the checkers program. Users should run this file to start the game.
- **Board.py** [3]: The Board.py is for the GUI of the checkers game. It represents to the user the basic checkers board with white&red checkers tokens.
- **Constants.py** [5]: Constants.py, as the name suggests, various constant values are stored.
- **Game.py** [7]: The Game.py is a file for handling the checkers game as a whole. The process of updating the location of the checkers' token that the moved through the mouse and determining whether it is a valid move or not under the game rules takes place inside this file.
- **Token.py** [11]: The Token.py is a file related to checkers tokens that provide basic position and color values (This color value allows the program to recognize which player uses this token.) of tokens for the game.
- **Search.py** [10]: The Search.py file contains logic for AI players. The AI player uses the minimax algorithm with Alpha-Beta pruning determine the optimal movement here.
- **AI_Move.py** [2]: The AI_Move.py is a file that calculate the valid move of AI player only.
- **Men.py** [9]: The Men.py is a file that suggests the valid men token's move to the program. It determines whether the valid capturing move is possible or not, and also determines whether it is possible to convert men token as a king token from a certain position.
- **King.py** [8]: The King.py is a file that suggest the valid king token's move (added backward move to men token move.) to the program. It determines whether valid capturing move is possible or not.
- **Display.py** [6]: The Display.py is a file that displays the interface of game status to the user. It draw highlights the tiles which is a valid position to move on for hint to the player.
- **Button.py** [4]: The Button.py is a file for drawing a pressable button on the screen.

2. Compilation and Run

This Checkers game is written in Python (pygame 2.0.1 (SDL 2.0.14, Python 3.7.9)) with visual studio code. (https://code.visualstudio.com/) So, please install the **1. visualstudio code** and **2. pygame** (only for GUI part) properly to run this program.
I tested this program with 5 Lab's computers and confirmed it working well. Also, found out most of Lab's computers already have pygame. However, if you use a computer that doesn't have pygame, then, just open the command prompt and put the command as shown below:

Install pygame with pygame command:

```
pip install pygame
```

Different way to install pygame to the computer:

Or, if the command does not work well, just unzip the site-packages.zip folder (that includes two files, 'pygame', 'pygame-2.0.1.dist-info') which is included in the assessment file, and put it to the correct path where your computer store all of your external python libraries. (In Lab computer case, the path is Figure 2)

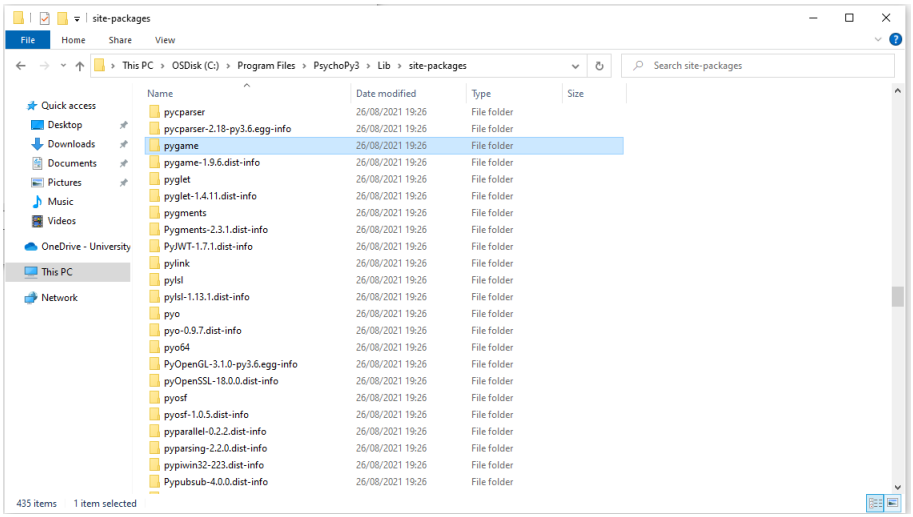

Figure 2: Lab's computer proper path of pygame

Install requirement extension pack of VScode:

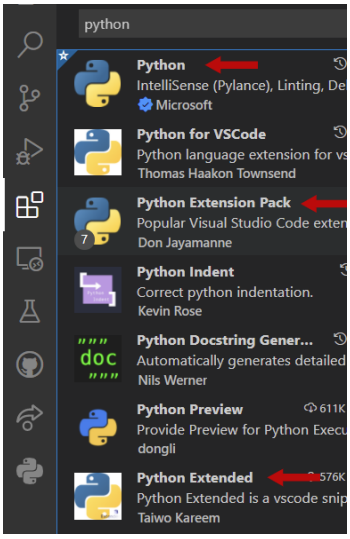

Figure 3: Visual Studio code Extension packs Setting

After installing the VScode through following site (https://code.visualstudio.com/), the user should install the proper extension packs. Requirement extension packs of this program are (Figure 3):

1. Python 2. Python Extended 3. Python Extension Pack.

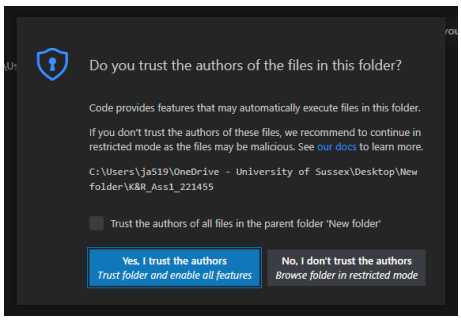

Figure 4: Screenshot of VScode

Then, just press the 'trust the authors' button (Figure 4) for running this program.

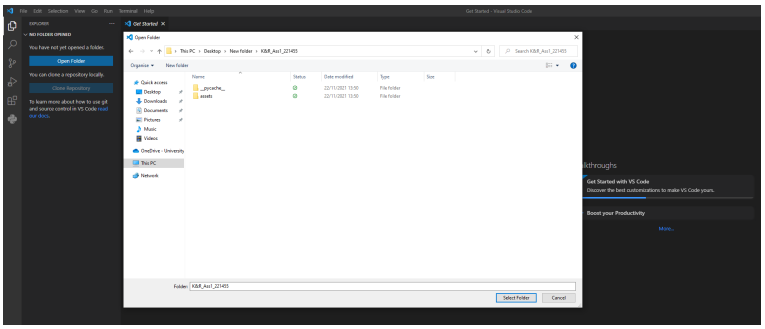

Figure 5: Open the correct program folder to run checkers

It is important to click the 'file' button above and open the right folder to run the program. Since the included assets should be recognized to the computer, 'K&R_Ass1_221455' folder needs to open exactly on the VScode. (Figure 5)

1. Install visual studio code (https://code.visualstudio.com/) > 2. Install proper extension of visual studio code > 3. Install pygame > 4. Run main.py through VSCode with proper folder open

All information is briefly written on '**README.txt**' which includes in the assessment folder.

Screenshot of testing program on Lab's computer with VScode:

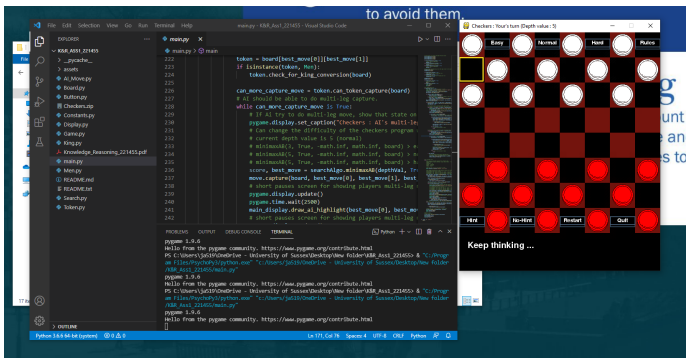

Figure 6: Testing program on Lab's computer with Visual Studio Code

3. Game Internals

3.1. Validation of moves

All legal moves of each token are stored as a list with multiple tuple objects through a successor function in which allows the token to move. The **calc_all_possible_moves()**[11, lines 49] function returns the all possible valid moves from the current token's position and the boolean value of whether capture move is possible (return True) or not (return False). During the player's turn is continuing with the while loop, this function calculates the all possible moves of the token that is clicked by the player and highlights the tile that possible to move on. When the player moves the token in a certain way, the program updates the board state and displays the results of calculating all legal moves from the token's new location on the screen again, recursively.

calc_all_possible_moves: (board) -> (tuple[possible move list, Literal[bool=False]] or tuple[capturing move list, Literal[bool=True]])

In addition, these groups of valid moves are obtained by AI_Move.py [2, lines 23], Game.py[7, lines 63], Men.py[9, lines 23], and King.py[8, lines 10] with proper functions. Among them, AI_Move.py[2, lines 14] file is exist only to calculate the AI's valid movements.

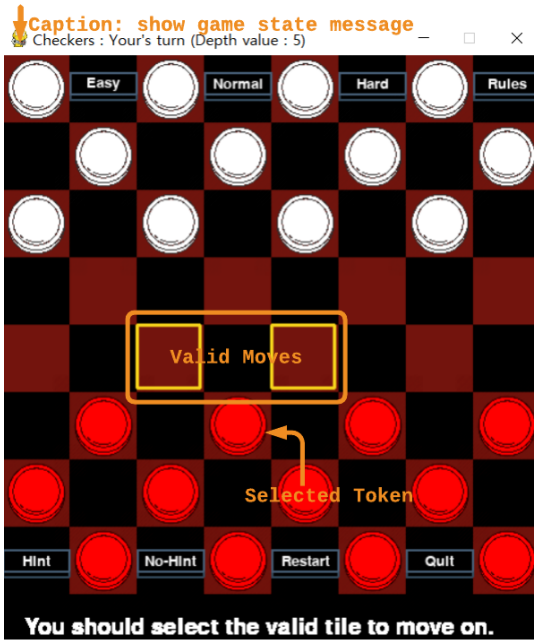

Figure 7: Board representation when selected the token

There are three different situations appears that can be considered when token's move.

- In the case the token does not exist in the tile where the player is trying to move into. **(Possible to move into. -> Valid move)**
- In the case the same color of token exist in the tile where the player is trying to move into. **(Impossible to move into. -> Invalid move)**
- In the case the opposite color of token exist in the tile where the player is trying to move into.**(Should keep looking on the diagonal tiles. -> Cannot judge yet.)**

- In the case the token does not exist in the back tile where the player is trying to move into. **(Possible to move into. -> Valid move)**

; It is a capturing move, therefore, should remove the token that is captured.

- In the case the same&opposite color of token exist in the back tile where the player is trying to move into. **(Impossible to move into. -> Invalid move)**

1. Non-capturing move - Men Token (Move forward)
2. Non-capturing move - King Token (Move forward&Move Backward)
3. Capturing move

3.2. Men token: Non-Capturing move

The men token can only move forward from the initial position. There are two types of movement that can take action. Non-capturing move and capturing movement. Non-capturing movement is simply a diagonal movement from one square to an adjacent square. Since only diagonal progress is possible, tokens do not place on black tiles in this program never in this program. (Figure 7) Men token's possible move store in the **Men.py[9]**&**Token.py[11]**.

3.3. King token: Non-Capturing move

The king token can move almost the same as men token, however, they are also able to move backward from the initial position. Men token become king token (**; king conversion**) when they reach to the king's rows through king_conversion_men() function in **Men.py[9, lines 35]** or capturing the opponent king token (**Regicide**). The regicide function in this program simply implemented as checking whether captured token is king or not with boolean value in **Men.py[9, lines 46]**. All King's token is distinguished from men token by the inside crown marked. King token's possible move store in the **King.py[8]**&**Token.py[11]**. Below(Figure 8) is the king token's non-capturing legal move in this checkers program:

Figure 8: King token's valid move

3.4. Capturing move

The move when the token "jumps" the enemy's token is called capturing-move. This is done diagonally as a non-capturing move and can only happen if the tile at the back is also open. In addition, this capturing-move can be repeated several times if possible. This means that a player can jump several times while capturing several tokens at once time. It is called multi-leg capture.

Below[1] pseudo code indicates the algorithm for calculating the possible moves:

Algorithm 1 Calculating the valid moves

```
1: Calculate_valid_moves (token's goal y position, token's goal x position, token's current
   y position, token's current x position)
2:
3: all valid moves ← empty list
4: all valid captures ← empty list
5: can_capture_the_token ← boolean
6:
7: while Player.turn() is True:
8:
9:     can_capture_the_token = False
10:
11:    #non-capturing move
12:    if is possible move available(current y position, current x position) :
13:        all valid moves.append(goal y position, goal x position)
14:    #capturing move
15:    if is capturing move available(current y position, current x position) :
16:        all valid captures.append(goal y position, goal x position)
17:        #In the case of all valid captures list is not empty
18:        can_capture_the_token ← True
19:
20:        return can_capture_the_token
21:
22:    if can_capture_the_token is True:
23:        return all valid captures ← Update the current state after move
24:    else:
25:        return all valid moves ← Update the current state after move
26:
27: end while
```

3.5. Forced capture

For calculating all valid moves, firstly, store the possible non-capturing move and possible capturing move from the current position as a tuple object to the each appropriate list. Afterward, if the list that storing the capturing move is not an empty list, make the boolean value as 'True' that checks whether capturing move is possible or not. If the boolean value indicates that capturing move is valid from an initial position, return capturing move list at the end of the function. Conversely, if the capturing move is an empty list, a non-capturing move list is

returned.[7, lines 83] This process allows the player to achieve the rule of **forced captures**. (Figure 9) Player has more than one capturing move to choose from, the player must do it.

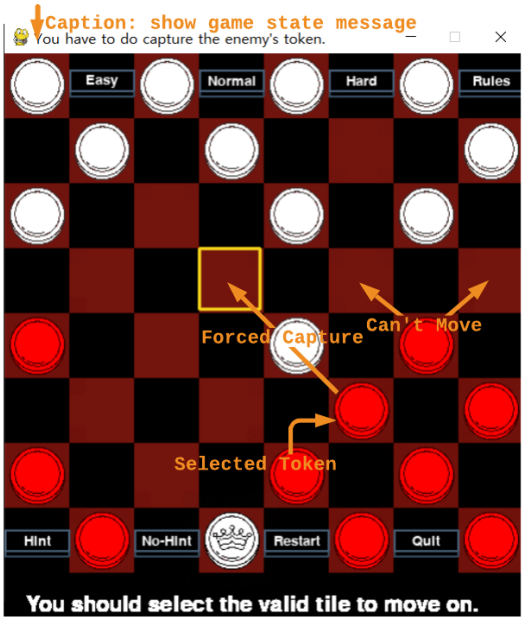

Figure 9: Capturing move

3.6. Multi-leg capture

In the case of taking a non-capturing move, changes turn after finishing the token's move and updating the new token's location information to the board.
In contrast, in the case of capturing move is taken, the program checks whether capturing move is possible again at the updated position after the first move. This steps for **multi-leg capturing moves**. If capturing move is possible again, repeat the same step recursively and return the new list of capturing move[1, lines 226], otherwise, switch the turns.

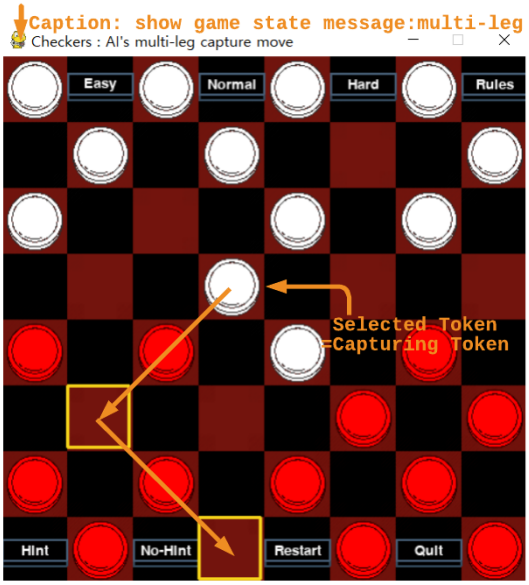

Figure 10: Multi-leg capturing moves of AI

In addition, the function pygame.time.wait(2500)[1, lines 240&244] for short pauses was added to the function of recursively calculating the valid capturing move so that the intermediate step can be confirmed when multi-leg capture is performed. The player can see the inform message on the caption during multi-leg capture is in progress. Also, the tile that is scheduled to proceed with the multi-capturing move, is highlighted and appears on the board sequencely. (Figure 10)

3.7. Search Algorithm: Heuristics

The heuristic function for executing the minimax algorithm was written based on two facts that can be generally considered when playing checkers.

1. Firstly, having more king tokens is advantageous in winning the game. So, token should try to locate on the king's rows. [10, lines 21]

2. Secondly, it is safe if the token is located in the leftmost or rightmost columns. Since if the token is located in that position, the opposing token cannot capture my token. So, token should try to locate on the leftmost or rightmost columns. [10, lines 41]

Heuristic	Score
Men (Token State)	15
King (Token State)	45
non-high_score_special_position (Position state)	1
high_score_special_position (Position state)	1.3

Table 1: Heuristic

$$HeuristicScore = TokenState \times PositionState \tag{1}$$

Therefore, heuristic scores are evaluated by simple multiplication based on the current state of the token (men: 15, King: 45) and the position (normal: 1, special-case: 1.3). [10, lines 61]

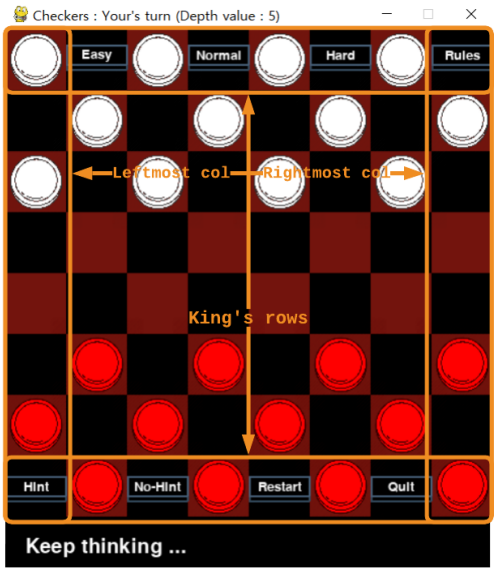

Figure 11: High Score position on the board

3.8. Search Algorithm: Minimax and Alpha Beta Pruning

The AI of the checkers program, which becomes the enemy player of the human player, has become a rational agent with the minimax algorithm optimized with alpha-beta pruning. At this time, alpha-beta pruning uses for discarding the sections of the search space that have proven to be unfavorable to each player. [10, lines 80]
The below pseudo code briefly explains the AI algorithm implemented in the program:

Algorithm 2 Calculating the valid moves

```
1: MinimaxAB (Depth, maxPlayer, alpha, beta, board_state)
2:
3: If max depth reached or game is end
4:     return score based on heuristic
5:
6: else:
7:     If maxPlayer is True: #maxPlayer turn
8:         Best score = -math.inf
9:         for each valid move for this player:
10:            token move
11:            score = minimax (depth - 1, minPlayer, alpha, beta)
12:            # Point of entry into the recursion
13:            undo move
14:            Best score = max(score, Best score)
15:            alpha = max(alpha, Best score)
16:            # MaxPlayer updates alpha
17:            If (alpha >= beta):
18:                break # beta cut-off
19:        return Best score
20:
21:    else:
22:        If maxPlayer is False: #minPlayer turn
23:            Least score = +math.inf
24:            for each valid move for this player:
25:                token move
26:                score = minimax (depth - 1, maxPlayer, alpha, beta)
27:                # Point of entry into the recursion
28:                undo move
29:                Least score = min(score, Least score)
30:                beta = min(beta, Least score)
31:                # MinPlayer updates beta
32:                If (alpha >= beta):
33:                    break # alpha cut-off
34:            return Least score
```

Each score is calculated through a heuristic function, and the valid moves of each token is obtained from the function of **calc_all_possible_moves()** that explains in section 3.1. During scoring each token move, undoing the move as it recursively during exploring the search tree. Thanks to the undoing process, the program can get the score of each movement with maintaining the token's initial position on the board that the player sees on the screen. (Token should not be located on the simulated tile when calculating the score on the board.)

4. Human-Computer Interface

4.1. Design the checkers board

This checkers program has a board composed of 8 columns & 8 rows. Twelve, flat disc-like tokens for each player are placed on the dark-red tiles in the manner indicated in the diagram below (Figure 12). In this program, a human player uses darker colored tokens(RED), and an AI player uses brighter colored tokens(White). The token is drawn by Display.py [6, lines 60] on the board. In addition, the position where the token should be located informed by the board state represent as below:

Token exist on the board position(x, y):
board[token_position_y][token_position_x] != None
Token not exist on the board position(x, y):
board[token_position_y][token_position_x] = None

When the human player hovers the mouse over the board, a yellow square-shaped border is drawn to the tile indicate that it has been highlighted. If the player clicks one specific token, the tiles for possible moves of the token are highlighted. Plus, the board shows a token that is selected by AI player and its goal tile with a highlight. The function of the drawing highlights and checkers board is described on the Display.py[6, lines 41&lines 99].

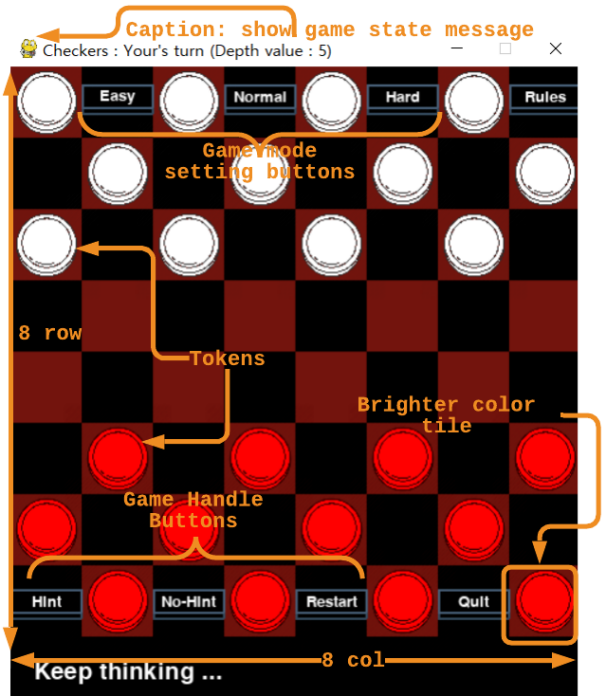

Figure 12: Checkers game board

4.2. Board state representation on the screen: Caption

In the caption at the top of the screen, typically, a message appears to distinguish whether it is the turn of AI or the turn of the human player. There are a total of four board presentations appearing in the caption:

1. A message of representing AI's turn

2. A message of representing a human player's turn (Figure 7)
3. A message of a human player has to do forced-captured move (Figure 9)
4. A message of AI doing multi-leg capturing move (Figure 10)

4.3. In-game help features: Highlight

The program highlights the currently available goal tiles of each token through the 3px yellow border to give a hint to the player. In other words, the player cannot move with tiles that have not been highlighted. These highlights are drawn written in a Display.py[6, lines 109] that deals with all elements displayed on the screen. This highlight hint option is adjustable by the user with the buttons[1, lines 95] on the bottom left labeled as 'Hint' and 'No-Hint'.

4.4. Display the checkers game rules: Button

Figure 13 indicates there is a button existing on the top-right corner labeled with 'Rules'. This is a button for outputting a brief **checkers rule** to the black blank window below with Button.py[4, lines 43]. If there are some user who has no knowledge of checkers games, he can read the basic rules for checkers games on the screen just by clicking the button. There are also 'restart' and 'quit' button on the bottom right side to handle the checkers game.

4.5. Different levels of verifiably effective AI cleverness: Button

Figure 13 indicates there are three different buttons on the top labeled with 'Easy', 'Normal', and 'Hard'.
The degree of the cleverness of AI in the minimax algorithm varies depending on the search depth. Currently, 5 is set as the default depth value in this program, which has a normal degree of cleverness. If this depth value is set to 3, the game becomes an easy mode, if it is set to 5 the game becomes a normal mode, and it becomes hard mode if it is set to 7.
This depth value[1, lines 217&236] of the minimax algorithm can be simply adjusted by user with the buttons[1, lines 34] on the screen above. Every time the user press the button, the depth value changes, and user can confirm the value on the caption. The larger depth value leads the minimaxAB algorithm to search and consider the positions of tokens in the more distant future.

Figure 13: Buttons on the screen

Appendices

A. Source code

A.1. main.py

```
# main.py
import sys
import pygame
from pygame.locals import *
import math
from Display import Display
from Board import Board
from Men import Men
from Token import Token
from Game import Game
from AI_Move import AI_Move
from Search import Search
from Constants import *
from Button import Button
from pygame import mixer


fps = int(FPS)

def main():
    """
    To run this checkers program, should run this main.py file.
    """
    pygame.init()

    fps_clock = pygame.time.Clock()

    # Set the "Checkers" title on the game interface
    pygame.display.set_caption('Checkers')

    main_board = Board()
    main_display = Display()

    difficultyButtons = [
        Button('Rules', 60, 20, (420, 20), 5),
        Button('Easy', 60, 20, (60, 20), 5),
        Button('Normal', 60, 20, (180, 20), 5),
        Button('Hard', 60, 20, (300, 20), 5),
    ]

    hintButtons = [
        Button('Hint', 60, 20, (0, 445), 5),
        Button('No-Hint', 60, 20, (120, 445), 5)
    ]

    playButtons = [
        Button('Restart', 60, 20, (240, 445), 5),
        Button('Quit', 60, 20, (360, 445), 5)
    ]

    # Initial value of the checkers game
```

```
    mouse_selected = False
    mouse_x = 0
    mouse_y = 0
    spotx = 0
    spoty = 0
    board = main_board.create_board()
    # Human player uses a WHITE token.
    # If you change 'WHITE' value as 'RED', the position is changed. (Player can handle the WHITE tokens that on the upside on game screen.)
    human_player = Game(RED, True)
    ai_player = Game(WHITE, False)
    # AI (WHITE player) uses Minimax-alpha beta algorithm for this game.
    searchAlgo = Search(ai_player, human_player, WHITE)
    move = AI_Move()
    depthVal = 5

    showHighlight = True

    # Loop for run the main game.
    while True:

        current_player = Game.select_player_with_turn(human_player, ai_player)
        main_display.update_board(board)

        # Draw buttons for setting the game difficulty level
        difficultyButtons[0].draw()
        difficultyButtons[1].draw()
        difficultyButtons[2].draw()
        difficultyButtons[3].draw()

        if difficultyButtons[1].pressed is True:
            # Easy - search 3 ahead move
            depthVal = 3
        elif difficultyButtons[2].pressed is True:
            # Normal - search 5 ahead move
            depthVal = 5
        elif difficultyButtons[3].pressed is True:
            # Hard - search 7 ahead move
            depthVal = 7

        # Draw buttons for highliting or not the legal tiles for the user as a hint
        hintButtons[0].draw()
        hintButtons[1].draw()

        if hintButtons[0].pressed is True:
            # Give user the hint of legal goal tile
            showHighlight = True
        if hintButtons[1].pressed is True:
            # Do not Give user the hint of legal goal tile
            showHighlight = False

        playButtons[0].draw()
        playButtons[1].draw()

        if playButtons[0].pressed is True:
            # Re-start the game
            main()
```

```
        if playButtons[1].pressed is True:
            # Quit the game
            pygame.quit()
            sys.exit()

        # For human player turn.
        if human_player.turn is True:
            # Show the who's turn on the caption of screen
            pygame.display.set_caption("Checkers : Your's turn (Depth value : %d)" % depthVal)
            # checks whether some token is captured or not.
            is_captured = human_player.is_token_captured(board)
            main_display.check_game_is_end()

            for event in pygame.event.get():
                if event.type == pygame.MOUSEMOTION:
                    mouse_y, mouse_x = event.pos
                # Check the token which is selected by mouse or not and confirm the mouse position.
                if event.type == pygame.MOUSEBUTTONUP:
                    spoty, spotx = main_display.draw_spot_selected(board, event.pos[0], event.pos[1])
                    mouse_selected = True
            main_display.highlight_while_thinking(board, main_display, mouse_selected, mouse_y, mouse_x)
            token = board[spoty][spotx]

            if isinstance(token, Token) and token.color == human_player.color and mouse_selected is True:
                possible_moves, capture = token.calc_all_possible_moves(board)
                has_captured = False

                # This While loop for controlling the capture moves.
                # If there are no capture move exist, this loop is just skipped.
                while any(True in sublist for sublist in possible_moves) and capture is True:
                    pygame.display.set_caption("You have to do capture the enemy's token.")
                    # show users the possible captures through highlight.
                    if showHighlight is True:
                        main_display.highlight_possible_moves(possible_moves)
                    event = pygame.event.wait()
                    main_display.check_game_is_end()

                    if event.type == pygame.MOUSEBUTTONUP:
                        # if the mouse is on top of the token, token is marked.
                        tile_to_move_y, tile_to_move_x = main_display.draw_spot_selected(board, event.pos[0], event.pos[1])

                        if possible_moves[tile_to_move_y][tile_to_move_x] is True:
                            main_display.capture_token_animation(board, tile_to_move_y, tile_to_move_x, token.color, spoty, spotx)
                            token.capture_token(board, tile_to_move_y, tile_to_move_x)
```

```
                            spoty, spotx = tile_to_move_y, tile_to_move_x

                            if isinstance(token, Men):
                                token.check_for_king_conversion(board)

                            possible_moves, capture = token.
calc_all_possible_moves(board)
                            has_captured = True

                        elif has_captured is False:
                            # return the selection of token.
                            break

                # This While loop for controlling the non-capture moves.
                while any(True in sublist for sublist in possible_moves) and
 not has_captured and not is_captured:
                    if showHighlight is True:
                        main_display.highlight_possible_moves(possible_moves
)
                    event = pygame.event.wait()
                    main_display.check_game_is_end()

                    # if the mouse is on top of the token, token is marked.
                    if event.type == pygame.MOUSEBUTTONUP:
                        tile_to_move_y, tile_to_move_x = main_display.
draw_spot_selected(board, event.pos[0], event.pos[1])

                        if possible_moves[tile_to_move_y][tile_to_move_x] is
 True:
                            main_display.move_token_animation(board,
tile_to_move_y, tile_to_move_x,
                                                              token.color,
spoty, spotx)
                            token.make_move(board, tile_to_move_y,
tile_to_move_x)
                            spoty, spotx = tile_to_move_y, tile_to_move_x

                            if isinstance(token, Men):
                                token.check_for_king_conversion(board)

                            # End the current turn.
                            human_player.changes_turns(human_player,
ai_player)
                            mouse_y, mouse_x = event.pos
                            possible_moves = [[]]
                        else:
                            # return the selection of token.
                            break

                if has_captured:
                    # the turn should be end after capturing move.
                    # change the turn - from human player to AI player.
                    human_player.changes_turns(human_player, ai_player)
                    mouse_y, mouse_x = event.pos


        # For AI player turn
        else:
            # Uses minimax Alpha-Beta algorithm for checkers game
```

```
202             # minimaxAB(self, depth, maxPlayer, alpha, beta, board) - from
    Search.py
203             # depth : 5
204             # maxPlayer : AI
205             # alpha : -infinite
206             # beta : +infinite
207             # board : board
208 
209             # Show the who's turn on the caption of screen
210             pygame.display.set_caption("Checkers : AI's turn")
211 
212             # Can change the difficulty of the checkers program with handle
    depth value.
213             # current depth value is 5 (normal)
214             # minimaxAB(3, True, -math.inf, math.inf, board) > easy mode
215             # minimaxAB(5, True, -math.inf, math.inf, board) > normal mode
216             # minimaxAB(5, True, -math.inf, math.inf, board) > hard mode
217             score, best_move = searchAlgo.minimaxAB(depthVal, True, -math.
    inf, math.inf, board)
218             if best_move[4] is not None:
219                 move.capture(board, best_move[0], best_move[1], best_move
    [2], best_move[3], best_move[4])
220                 main_display.draw_ai_highlight(best_move[0], best_move[1])
221 
222                 token = board[best_move[0]][best_move[1]]
223                 if isinstance(token, Men):
224                     token.check_for_king_conversion(board)
225 
226                 can_more_capture_move = token.can_token_capture(board)
227                 # AI should be able to do multi-leg capture.
228                 while can_more_capture_move is True:
229                     # If Ai try to do multi-leg move, show that state on the
     caption.
230                     pygame.display.set_caption("Checkers : AI's multi-leg
    capture move")
231                     # Can change the difficulty of the checkers program with
     handle depth value by click the button.
232                     # current depth value is 5 (normal)
233                     # minimaxAB(3, True, -math.inf, math.inf, board) > easy
    mode
234                     # minimaxAB(5, True, -math.inf, math.inf, board) >
    normal mode
235                     # minimaxAB(5, True, -math.inf, math.inf, board) > hard
    mode
236                     score, best_move = searchAlgo.minimaxAB(depthVal, True,
    math.inf, -math.inf, board)
237                     move.capture(board, best_move[0], best_move[1],
    best_move[2], best_move[3], best_move[4])
238                     # short pauses screen for showing players multi-leg
    moves tiles.
239                     pygame.display.update()
240                     pygame.time.wait(2500)
241                     main_display.draw_ai_highlight(best_move[0], best_move
    [1])
242                     # short pauses screen for showing players multi-leg
    moves tiles.
243                     pygame.display.update()
244                     pygame.time.wait(2500)
245                     token = board[best_move[0]][best_move[1]]
```

```
                    if isinstance(token, Men):
                        token.check_for_king_conversion(board)
                    can_more_capture_move = token.can_token_capture(board)
            # non-capture touple value that does not have any fifth item.
            else:
                move.move(board, best_move[0], best_move[1], best_move[2],
    best_move[3])
                main_display.draw_ai_highlight(best_move[0], best_move[1])
                token = board[best_move[0]][best_move[1]]

                if isinstance(token, Men):
                    token.check_for_king_conversion(board)

            # Change the turn - From AI player to human player.
            ai_player.changes_turns(human_player, ai_player)
            pygame.display.update()
            pygame.time.wait(300)

        # Redraw screen.
        # Wait a clock tick.
        current_player.check_who_win(board, main_display)
        mouse_selected = False
        pygame.display.update()
        fps_clock.tick()


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

Listing 1: main.py

A.2. AI_Move.py

```
# AI_Move.py
from Men import Men
from King import King



class AI_Move:
    # Class for calculating the valid move of AI player.
    def __init__(self):
        pass

    # Confirm that if there are possible capture moves exist or not on a
    given tile.
    # If there are possible capture moves exist, calculate the position of
    the captured token and capture token.
    def capture(self, board, goal_y, goal_x, posy, posx, captured_token):
         # remove the token on the ([captured_token_position_y][
    captured_token_position_x]) tile which is captured.
        board[captured_token.posy][captured_token.posx] = None
        board[posy][posx], board[goal_y][goal_x] = board[goal_y][goal_x],
    board[posy][posx]
        # Update the position of capture token.
        board[goal_y][goal_x].posy = goal_y
        board[goal_y][goal_x].posx = goal_x

```

```
    # Calculate the position of the token after move.
    def move(self, board, goal_y, goal_x, posy, posx):
        board[posy][posx], board[goal_y][goal_x] = board[goal_y][goal_x], board[posy][posx]
        # Update the position of current token after valid move.
        board[goal_y][goal_x].posy = goal_y
        board[goal_y][goal_x].posx = goal_x

    # Memorize the position of token before simulating move with minimax-alphabeta algorithm.
    def undo_move(self, board, goal_y, goal_x, posy, posx):
        board[posy][posx], board[goal_y][goal_x] = board[goal_y][goal_x], board[posy][posx]
        board[posy][posx].posy = posy
        board[posy][posx].posx = posx

    # Memorize the position of token before simulating capture-move with minimax-alphabeta algorithm.
    def undo_capture(self, board, goal_y, goal_x, posy, posx, captured_token):
          board[captured_token.posy][captured_token.posx] = captured_token
          board[posy][posx], board[goal_y][goal_x] = board[goal_y][goal_x], board[posy][posx]
          board[posy][posx].posy = posy
          board[posy][posx].posx = posx
```

Listing 2: AI_Move.py

A.3. Board.py

```
# Board.py
import Men
from Constants import *



class Board:
    # Class for initial Board of checkers game.
    def __init__(self):
        self.size_board = int(SIZE_BOARD)
    # Draw a checkers 2D board with 8 col and 8 rows.
    def create_board(self):
        # Make an empty 2D list & Men token objects on the board.
        new_board = [[None for j in range(self.size_board)] for i in range(self.size_board)]
        for i in range(self.size_board // 2 - 1):
            for j in range(0, self.size_board, 2):
                # Give the WHITE (for ai) tokens for initial state of checkers game.
                new_board[i][j + i % 2] = Men.Men(i, j + i % 2, WHITE)
        for i in range(self.size_board // 2 + 1, self.size_board):
            for j in range(0, self.size_board, 2):
                # Give the RED (for player) tokens for initial state of checkers game.
                new_board[i][j + i % 2] = Men.Men(i, j + i % 2, RED)
        # Return the 2D board structure data and positions of Men token objects.
```

```
        return new_board
```

Listing 3: Board.py

A.4. Button.py

```
# Button.py
import pygame, sys
from Constants import *
from pygame.locals import *



screen = pygame.display.set_mode((500,500))
class Button:
    # Class for making a GUI button on the screen.
    def __init__(self, message, width, height, pos, elevation):
      #Core attributes

      self.pressed = False
      self.elevation = elevation
      self.dynamic_elecation = elevation
      self.original_y_pos = pos[1]

      # top rectangle
      self.top_rect = pygame.Rect(pos, (width, height))
      self.top_color = DARKBLUE

    # bottom rectangle
      self.bottom_rect = pygame.Rect(pos, (width, height))
      self.bottom_color = DEEPBLUE
    # message
      self.message_surf = pygame.font.Font(None, 20).render(message, True, WHITE)
      self.message_rect = self.message_surf.get_rect(center = self.top_rect.center)

    def draw(self):
    # elevation logic
      self.top_rect.y = self.original_y_pos - self.dynamic_elecation
      self.message_rect.center = self.top_rect.center

      self.bottom_rect.midtop = self.top_rect.midtop
      self.bottom_rect.height = self.top_rect.height + self.dynamic_elecation

      pygame.draw.rect(screen, DEEPBLUE, self.bottom_rect, 2)
      pygame.draw.rect(screen, DARKBLUE, self.top_rect, 2)
      screen.blit(self.message_surf, self.message_rect)
      self.show_rule()

    def show_rule(self):
        self.font = pygame.font.Font(None, 20)
        mouse_pos = pygame.mouse.get_pos()
        state_message_1 = self.font.render("Rule 1. You can handle your men token with a diagonal move forward.", True, WHITE)
        state_message_2 = self.font.render("Rule 2. You can handle your king token with a backward move also.", True, WHITE)
```

```
48          state_message_3 = self.font.render("Rule 3. If you can capture an
    enemy token, then you have to do so.", True, WHITE)
49          state_message_4 = self.font.render("Rule 4. You can upgraded men
    token as king token to reach king cols.", True, WHITE)
50          if self.top_rect.collidepoint(mouse_pos):
51              self.top_color = BRIGHTRED
52              if pygame.mouse.get_pressed()[0]:
53                  self.dynamic_elecation = 0
54                  self.pressed = True
55              else:
56                  self.dynamic_elecation = self.elevation
57
58                  if self.pressed == True:
59                      screen.blit(state_message_1, (20, 490))
60                      screen.blit(state_message_2, (20, 510))
61                      screen.blit(state_message_3, (20, 530))
62                      screen.blit(state_message_4, (20, 550))
63                      pygame.display.update()
64                      event = pygame.event.wait()
65                      self.pressed = False
66          else:
67              self.dynamic_elecation = self.elevation
68              self.top_color = DARKBLUE
```

Listing 4: Button.py

A.5. Constants.py

```
1  # Constants.py
2  import pygame
3
4
5
6  # Contants value for checkers game
7
8  # Interface size (width and height)
9  WIDTH, HEIGHT = 480, 570
10 # value for 8x8 2D checkers game board
11 ROWS, COLS = 8, 8
12 SIZE_BOARD = 8
13 SIZE_TILE = 60
14
15 FPS = 30
16
17 # Set the color with RGB value
18 RED =        (255, 0, 0)
19 WHITE =      (255, 255, 255)
20 BLACK =      (0, 0, 0)
21 DARKRED =    (115, 20, 13)
22 BLUE =       (0, 0, 255)
23 YELLOW =     (249, 215, 28)
24 DARKBLUE =   (71, 95, 119)
25 DEEPBLUE =   (53, 75, 94)
26 BRIGHTRED =  (215, 75, 75)
27
28 FONT_SIZE = 30
29
30 # Input the png image to the program
```

```
CROWN = pygame.transform.scale(pygame.image.load('assets/crown.png'), (90, 90))
REDMEN = pygame.transform.scale(pygame.image.load('assets/redMen.png'), (90, 90))
WHITEMEN = pygame.transform.scale(pygame.image.load('assets/whiteMen.png'), (90, 90))
```

Listing 5: Constants.py

A.6. Display.py

```
# Display.py
import pygame
from pygame.locals import *
import sys
from Token import Token
from Constants import *
import Men
import King
from Game import *



class Display:
    # Class for user interface for display game status for user.
    # Initialize of each value from Constants.py.
    def __init__(self):
        self.size_board = int(SIZE_BOARD)
        self.size_tile = int(SIZE_TILE)
        self.window_width = int(WIDTH)
        self.window_height = int(HEIGHT)
        self.surface = pygame.display.set_mode((self.window_width, self.window_height))
        self.margin_x = int((self.window_height - (self.size_board * self.size_tile)) / 2)
        self.margin_y = int((self.window_width - (self.size_board * self.size_tile)) / 2)
        self.font_size = int(FONT_SIZE)
        self.font = pygame.font.Font(None, self.font_size)

    # Calculate the top-left corner positions (x,y) of given tile.
    def calc_top_left_corner(self, tile_y, tile_x):
        # tile_y: value of Board vertical position, same as row number in board 2D-list.
        # tile_x: value of Board horizontal position, same as column number in board 2D-list.
        top_left_x_pos = self.margin_y + (self.size_tile * tile_x)
        top_left_y_pos = self.size_tile * tile_y
        return (top_left_y_pos, top_left_x_pos)

    # Draw the square tiles for checkers game with pygame.draw.rect func.
    def draw_tile(self, tile_y, tile_x):
        top_left_y_pos, top_left_x_pos = self.calc_top_left_corner(tile_y, tile_x)
        pygame.draw.rect(self.surface, DARKRED, (top_left_x_pos, top_left_y_pos, self.size_tile, self.size_tile))

    # Draw empty board for checkers games.
```

```
41      def draw_empty_board(self):
42          # Draw black colored base-board.
43          self.surface.fill(BLACK)
44          self.board_dim = self.size_board * self.size_tile
45          # (0, 0) position is the start position of board and it is top-left
    on the screen.
46          top_left_y_pos, top_left_x_pos = self.calc_top_left_corner(0, 0)
47          # Draw black tile on white base-board which composed the checkers
    board.
48          pygame.draw.rect(self.surface, BLACK, (top_left_x_pos,
    top_left_y_pos, self.board_dim, self.board_dim))
49
50          count = 0
51          for tile_y in range(self.size_board):
52              for tile_x in range(self.size_board):
53                  if count % 2 == 0:
54                      self.draw_tile(tile_y, tile_x)
55                  count += 1
56              count += 1
57
58      # Draw men token for checkers game.
59      # Each men token object in which is drawn with .blit func on given tile.
60      def draw_men(self, tile_y, tile_x, color):
61          top_left_y_pos, top_left_x_pos = self.calc_top_left_corner(tile_y,
    tile_x)
62          if (color is WHITE):
63              self.surface.blit(WHITEMEN, (top_left_x_pos - 15, top_left_y_pos
     - 15))
64          else:
65              self.surface.blit(REDMEN, (top_left_x_pos - 15, top_left_y_pos -
     15))
66
67      # Mark King token with crown inside the normal token for distinguishing
    from the men token.
68      def draw_king_mark(self, tile_y, tile_x):
69          top_left_y_pos, top_left_x_pos = self.calc_top_left_corner(tile_y,
    tile_x)
70          # Draw the crown inside the King token with .blit func on the token.
71          self.surface.blit(CROWN, (top_left_x_pos - 15, top_left_y_pos - 15))
72
73      # Set the each token to the tiles on the board.
74      def tokens_on_board(self, board):
75          for tile_y in range(len(board)):
76              for tile_x in range(len(board[0])):
77                  # Set the men tokens on the board.
78                  if isinstance(board[tile_y][tile_x], Men.Men):
79                      self.draw_men(tile_y, tile_x, board[tile_y][tile_x].
    calc_color())
80                  # Set the king tokens on the board.
81                  elif isinstance(board[tile_y][tile_x], King.King):
82                      self.draw_men(tile_y, tile_x, board[tile_y][tile_x].
    calc_color())
83                      self.draw_king_mark(tile_y, tile_x)
84
85      # In the case of a token is clicked(selected) through a mouse,
86      # a spot is drawn on that for distinguishing it from other tokens.
87      def draw_spot_selected(self, board, pos_x_mouse, pos_y_mouse):
88          for tile_y in range(len(board)):
89              for tile_x in range(len(board)):
```

```
90                  top_left_y_pos, top_left_x_pos = self.calc_top_left_corner(
    tile_y, tile_x)
91                  # Show the spot on it with pygame.Rect() func.
92                  tile_Rect = pygame.Rect(top_left_x_pos, top_left_y_pos, self
    .size_tile, self.size_tile)
93                  # Change the pixel(x, y) positions to the board(x, y)
    positions (e.g. board: board data structure)
94                  if tile_Rect.collidepoint(pos_x_mouse, pos_y_mouse):
95                      return tile_y, tile_x
96          return None, None
97
98      # Draw the highlight on the tile which selected by the player with
    yellow square outlines.
99      def draw_player_highlight(self, tile_y, tile_x):
100         top_left_y_pos, top_left_x_pos = self.calc_top_left_corner(tile_y,
    tile_x)
101         pygame.draw.rect(self.surface, YELLOW, (top_left_x_pos,
    top_left_y_pos, self.size_tile - 3, self.size_tile - 3), 3)
102
103     # Draw the highlight on the tile which selected by the ai with yellow
    square outlines.
104     def draw_ai_highlight(self, tile_y, tile_x):
105         top_left_y_pos, top_left_x_pos = self.calc_top_left_corner(tile_y,
    tile_x)
106         pygame.draw.rect(self.surface, YELLOW, (top_left_x_pos,
    top_left_y_pos, self.size_tile - 3, self.size_tile - 3), 3)
107
108     # Draw the highlight on the tile where the token can move on with yellow
     square border.
109     def highlight_possible_moves(self, possible_moves):
110         for tile_y in range(len(possible_moves)):
111             for tile_x in range(len(possible_moves)):
112                 # Check the it is valid move or not.
113                 if possible_moves[tile_y][tile_x] is True:
114                     self.draw_player_highlight(tile_y, tile_x)
115                 else:
116                     state_message = self.font.render("You should select the
    valid tile to move on.", True, WHITE)
117                     self.surface.blit(state_message, (20, 500))
118
119         # Update the current state to the screen.
120         pygame.display.update()
121
122     # Update the board state.
123     def update_board(self, board):
124         # Call empty board.
125         self.draw_empty_board()
126         # Call all tokens on the board currently.
127         self.tokens_on_board(board)
128
129     def move_token_animation(self, board, tile_y, tile_x, color, posy, posx)
    :
130         """
131         : board: board data structure
132         : tile_y: Board vertical position at move's goal
133         : tile_x: Board horizontal position at move's goal
134         : color: color of given men
135         : posy: The vertical(y) pos value of token's current position on the
     board.
```

```
136         : posx: The horizontal(x) pos value of token's current position on
    the board.
137         """
138         self.draw_men(tile_y, tile_x, color)
139         self.draw_tile(posy, posx)
140         if isinstance(board[posy][posx], King.King):
141             self.draw_king_mark(tile_y, tile_x)
142 
143     def capture_token_animation(self, board, tile_y, tile_x, color, posy,
    posx):
144         captured_token_x = int((tile_x + posx) * 0.5)
145         if tile_y < posy:
146             captured_token_y = posy - 1
147         else:
148             captured_token_y = posy + 1
149         self.draw_men(tile_y, tile_x, color)
150         # If the token is King token, the token should have the king mark. (
    have crown inside the token)
151         if isinstance(board[posy][posx], King.King):
152             self.draw_king_mark(tile_y, tile_x)
153         self.draw_tile(posy, posx)
154         self.draw_tile(captured_token_y, captured_token_x)
155         pygame.display.update()
156 
157     # Short interface for checkers game end with winner message
158     def show_who_win(self, color):
159         myfont = pygame.font.SysFont(None, 30)
160         end_message = myfont.render("Checkers Game End!", True, BLACK)
161         pygame.display.set_caption("Game End")
162         if(color is RED):
163             game_outcome_message = myfont.render("Congratulations! You win!"
    , True, BLACK)
164         else:
165             game_outcome_message = myfont.render("   Try Again! AI win!",
    True, BLACK)
166         while True:
167             # Fill the interface with the winner's color (Player - RED / AI
    - WHITE)
168             self.surface.fill(color)
169             for event in pygame.event.get():
170                 if event.type == pygame.QUIT:
171                     pygame.quit()
172                     sys.exit()
173             self.surface.blit(end_message, (140, 130))
174             self.surface.blit(game_outcome_message, (140, 230))
175             pygame.display.update()
176             pygame.time.wait(3000)
177             pygame.quit()
178             sys.exit()
179 
180     # Shut down the game if escape keys or quit event input to the game.
181     def check_game_is_end(self):
182         for event in pygame.event.get():
183             if event.type == pygame.QUIT:
184                 self.shut_down()
185             pygame.event.post(event)
186         for event in pygame.event.get(pygame.KEYUP):
187             if event.key == pygame.K_ESCAPE:
188                 self.shut_down()
```

```
            pygame.event.post(event)

    # Shut down the program.
    def shut_down(self):
        pygame.quit()
        sys.exit()

    # If player keep hovering, just keep the highlight square to tiles that not selected yet on the board.
    def highlight_while_thinking(self, board, display, mouse_selected, mouse_y, mouse_x):
        # Check the tile is selected or not.
        if mouse_selected == False:
            tile_y, tile_x = display.draw_spot_selected(board, mouse_y, mouse_x)
            state_message = self.font.render("Keep thinking ...", True, WHITE)
            self.surface.blit(state_message, (20, 500))
            if tile_y != None and tile_x != None:
                display.draw_player_highlight(tile_y, tile_x)
```

Listing 6: Display.py

A.7. Game.py

```
# Game.py
from Constants import *
import Men
from Token import Token
from pygame import mixer


class Game:
    # Class for handling the general rules for checkers game.
    def __init__(self, color, turn):
        self.turn = turn
        self.color = color

    # Confirm the each player's turn.
    def calc_player_turn(self):
        return self.turn

    # Check the who (player or ai) win the checkers game.
    # After confirming, this checkers game show the screen with a message containing who is the winner.
    def check_who_win(self, board, main_interface):
        current_board = [y for x in board for y in x]
        tokens = [token for token in current_board if isinstance(token, Token) and token.calc_color() != self.color]
        # There are two goal state exist for checkers game
        # 1) Human player has zero checkers token left.
        # 2) AI player has zero checkers token left.
        if not tokens:
            main_interface.show_who_win(self.color)

    # Look at the board and check there are any player's tokens is captured or not.
    def is_token_captured(self, board):
```

```
        # list for captures tokens.
        all_captures = []
        for tile_x in range(len(board)):
            for tile_y in range(len(board)):
                if isinstance(board[tile_y][tile_x], Token) and board[tile_y
][tile_x].calc_color() == self.color:
                    possible_captures = board[tile_y][tile_x].
calc_possible_captures(board)
                    current_possible_captures = [y for x in
possible_captures for y in x]
                    all_captures.append(current_possible_captures)
        if any(True in sublist for sublist in all_captures):
            # case - Captured tokens exist
            return True
        else:
            # case - No captured tokens
            return False

    @classmethod
    def select_player_with_turn(cls, player1, player2):
        players = (player1, player2)
        for player in players:
            if player.turn is True:
                return player

    # Convert the turns of game.
    def changes_turns(self, player1, player2):
        players = [player1, player2]
        # Sound effect for placing the token
        mixer.music.load('assets/tokenPlaced.wav')
        for player in players:
            player.turn = not player.turn
            mixer.music.play()

    # Calculate the valid moves of tokens.
    def calc_moves_of_token(self, board, posy, posx):
        token_captures = []
        token_moves = []
        possible_moves, capture = board[posy][posx].calc_all_possible_moves(
board)
        for x_move in range(len(possible_moves)):
            for y_move in range(len(possible_moves)):
                if possible_moves[y_move][x_move] is True and capture is
True:
                    captured_token_x = int((posx + x_move) / 2)
                    if y_move < posy:
                        captured_token_y = posy - 1
                    else:
                        captured_token_y = posy + 1

                    capture_positions = (y_move, x_move, posy, posx,
board[captured_token_y][captured_token_x])
                    token_captures.append(capture_positions)

                elif possible_moves[y_move][x_move] is True and capture is
False:
                    move_positions = (y_move, x_move, posy, posx, None)
                    token_moves.append(move_positions)
        # If there are no valid capturing move exist, they just should move
```

```
    the token in valid way.
83          if not token_captures:
84              capture = False
85              return token_moves, capture
86          # If there are valid capturing move exist, they have to do so.
87          # ; Forced Capture
88          else:
89              capture = True
90              return token_captures, capture
91
92      # Calculate the all valid players moves; capturing moves & not capturing
        moves
93      def calc_player(self, board):
94          player_captures = []
95          player_moves = []
96          for tile_x in range(len(board)):
97              for tile_y in range(len(board)):
98                  if isinstance(board[tile_y][tile_x], Token) and board[tile_y
    ][tile_x].calc_color() == self.color:
99                      move_parameters, capture_move = self.calc_moves_of_token
    (board, tile_y, tile_x)
100                     if capture_move is True:
101                         player_captures.extend(move_parameters)
102                     else:
103                         player_moves.extend(move_parameters)
104
105         if not player_captures:
106             capture_move = False
107             return player_moves, capture_move
108         else:
109             capture_move = True
110             return player_captures, capture_move
```

Listing 7: Game.py

A.8. King.py

```
1  # King.py
2  from Constants import *
3  from Token import Token
4
5
6
7  class King(Token):
8      # Class for King token.
9      # For King token's advanced move different from men token.
10     def is_capture_available(self, board, tile_y, tile_x):
11         if tile_y < self.posy:
12             token_position_y = self.posy - 1
13         else:
14             token_position_y = self.posy + 1
15         token_position_x = int((self.posx + tile_x) / 2)
16
17         if abs(self.posx - tile_x) == 2 and abs(self.posy - tile_y) == 2 and
       not isinstance(board[tile_y][tile_x], Token):
18             if isinstance(board[token_position_y][token_position_x], Token)
    and board[token_position_y][token_position_x].calc_color() != self.color:
19                 return True
```

```
            else:
                return False
        else:
            return False

    # func for capturing move.
    def capture_token(self, board, tile_y, tile_x):
        if tile_y < self.posy:
            token_position_y = self.posy - 1
        else:
            token_position_y = self.posy + 1
        token_position_x = int((self.posx + tile_x) / 2)
        # remove the token on the ([token_position_y][token_position_x]) tile if that is captured.
        board[token_position_y][token_position_x] = None
        board[self.posy][self.posx], board[tile_y][tile_x] = board[tile_y][tile_x], board[self.posy][self.posx]
        self.posy = tile_y
        self.posx = tile_x

    # Confirm the it is valid move or not.
    def is_possible_move(self, board, tile_y, tile_x):
        if isinstance(board[tile_y][tile_x], Token):
            return False
        if abs(self.posy - tile_y) == 1 and abs(self.posx - tile_x) == 1:
            return True
        else:
            return False
```

Listing 8: King.py

A.9. Men.py

```
# Men.py
from Constants import *
from Token import Token
from King import King



class Men(Token):
    # Class for men token.
    # For men token's move different from King token.
    def __init__(self, posy, posx, color):
        # Constructor of the men class.
        super().__init__(posy, posx, color)
        # if the token's color is white(used by AI player), since white token's position is top side of the board, direction becomes -1 (go down)
        if self.color == WHITE:
            self.direction = -1
        # if the token's color is black(used by human player), since white token's position is down side of the board, direction becomes +1 (go up)
        else:
            self.direction = 1
        self.size_board = int(SIZE_BOARD)

    # Check that there are possible capture move exist or not on given tile.
    def is_capture_available(self, board, tile_y, tile_x):
```

```
24          token_position_y = self.posy - self.direction
25          token_position_x = int((self.posx + tile_x) / 2)
26          if abs(self.posx - tile_x) == 2 and self.posy - tile_y == 2 * self.
    direction and not isinstance(board[tile_y][tile_x], Token):
27              if isinstance(board[token_position_y][token_position_x], Token)
    and board[token_position_y][token_position_x].calc_color() != self.color:
28                  return True
29              else:
30                  return False
31          else:
32              return False
33
34      # If the men token reach some rule(ex. reach to the king col), men token
        becomes King token.
35      def king_conversion_men(self, board):
36          board[self.posy][self.posx] = King(self.posy, self.posx, self.color)
37
38      # Function for capturing the enemy's token.
39      # Update the Board data structure and token's position data (posy, posx)
        according to capturing move made.
40      def capture_token(self, board, tile_y, tile_x):
41          captured_token_position_y = self.posy - self.direction
42          captured_token_position_x = int((self.posx + tile_x) / 2)
43
44          # Regicide; if men token manages to capture a king, it is instantly
    become a king.
45          # check the captured token is King token or not.
46          if(isinstance(board[captured_token_position_y][
    captured_token_position_x], King)):
47              # remove the token on the ([captured_token_position_y][
    captured_token_position_x]) tile if that is captured.
48              board[captured_token_position_y][captured_token_position_x] =
    None
49              board[self.posy][self.posx], board[tile_y][tile_x] = board[
    tile_y][tile_x], board[self.posy][self.posx]
50              # Update the token's position data (posy, posx).
51              self.posy = tile_y
52              self.posx = tile_x
53              # Regicide; if men token captured King token, men token can be
    the King token.
54              self.king_conversion_men(board)
55
56          # non-Regicide
57          else:
58              # remove the token on the ([captured_token_position_y][
    captured_token_position_x]) tile if that is captured.
59              board[captured_token_position_y][captured_token_position_x] =
    None
60              # Update the token's position data (posy, posx).
61              board[self.posy][self.posx], board[tile_y][tile_x] = board[
    tile_y][tile_x], board[self.posy][self.posx]
62              self.posy = tile_y
63              self.posx = tile_x
64
65      # Check that the men token should become King token or not.
66      def check_for_king_conversion(self, board): #def
    check_for_king_conversion(self, board):
67          # if WHITE men token reach to the King col become a white king token
        with king_conversion_men func
```

```
68          if self.posy == self.size_board - 1 and self.color is WHITE:
69              self.king_conversion_men(board)
70          # if RED men token reach to the King col become a RED king token
    with king_conversion_men func
71          elif self.posy == 0 and self.color is RED:
72              self.king_conversion_men(board)
```

Listing 9: Men.py

A.10. Search.py

```
1  # Search.py
2  from Constants import *
3  import math
4  from Token import Token
5  from Men import Men
6  from AI_Move import AI_Move
7  from Game import Game
8
9
10
11 class Search:
12      # Class for search algorithm.
13      # This checkers game using the minimax algorithm for AI with
    optimization of alpha-beta pruning.
14      def __init__(self, ai, player, color):
15          self.color = color
16          self.player = player
17          self.ai = ai
18
19      # The more king tokens have, lead the more advantage to win the game.
20      # Therefore, a high score is given to move that try to reach the king
    row for becoming a king token.
21      def score_token_approach_king_cols(self, board, tile_y, tile_x):
22          # Set the score for each token.
23          men_score = 15
24          # King token should have more score than men token.
25          # Then, AI will try to make more and keep King token to maximise it'
    s score.
26          king_score = 45
27          high_score_special_position = 1
28
29          if board[tile_y][tile_x] is None:
30              return 0
31          # check the token reach the king row (The top row and the bottom row
    ) or not.
32          if tile_y == len(board) or tile_y == 0:
33              high_score_special_position = 1.3
34          if isinstance(board[tile_y][tile_x], Men):
35              return high_score_special_position * men_score
36          else:
37              return king_score
38
39      # If the token is located in the leftmost and rightmost col, it is not
    captured.
40      # Therefore, a high score is given to move to go to the vertical cols
    for keeping number of tokens to win.
41      def score_token_approach_vertical_cols(self, board, tile_y, tile_x):
```

```
        # Set the score for each token.
        men_score = 15
        # King token should have more score than men token.
        # Then, AI will try to make more and keep King token to maximise it's score.
        king_score = 45
        high_score_special_position = 1

        if board[tile_y][tile_x] is None:
            return 0
        # check the token reach the leftmost or rightmost col or not
        if tile_x == len(board) or tile_x == 0:
            high_score_special_position = 1.3

        if isinstance(board[tile_y][tile_x], Men):
            return high_score_special_position * men_score
        else:
            return king_score

    # Scoring the each token on the board.
    def scoring_board(self, board, depth):
        # initial state score is 0.
        score = 0
        # Calculating the token's score with considering their position
        # 1) if there are reach the king's row - can make more king tokens
        # 2) if ther are reach the leftmost or rightmost col - lead more safe for token
        for tile_x in range(len(board)):
            for tile_y in range(len(board)):
                if board[tile_y][tile_x] is not None:
                    if board[tile_y][tile_x].calc_color() == self.color:
                        score = depth + score + (self.score_token_approach_king_cols(board, tile_y, tile_x)
                                                 + self.score_token_approach_vertical_cols(board, tile_y, tile_x)) / 2
                    else:
                        score = score - depth - (self.score_token_approach_king_cols(board, tile_y, tile_x)
                                                 + self.score_token_approach_vertical_cols(board, tile_y, tile_x)) / 2
        # Return the calculated score
        return score

    # Implement the minimax algorithim that optimize with alpha-beta pruning.
    def minimaxAB(self, depth, maxPlayer, alpha, beta, board):
        best_move = 0
        move_ai = AI_Move()

        if depth == 0 or best_move is None:
            return self.scoring_board(board, depth), best_move
        # Considering the max-Player.
        if maxPlayer:
            best_score = -math.inf
            player, capture_move = self.ai.calc_player(board)
            score = -math.inf

            for leaf_Node in player:
                # Considering capture move.
```

```
94                 if capture_move:
95                     move_ai.capture(board, leaf_Node[0], leaf_Node[1],
    leaf_Node[2], leaf_Node[3], leaf_Node[4])
96                     # Considering capture move, especially, 'multi-leg moves
    '.
97                     can_more_capture_move = board[leaf_Node[0]][leaf_Node
    [1]].can_token_capture(board)
98                     if can_more_capture_move:
99                         score, player = self.minimaxAB(depth - 1, True,
    alpha, beta, board) # Point of entry into the recursion
100                        if score >= best_score:
101                            best_score = score
102                            best_move = leaf_Node
103                    # only once capture move
104                    else:
105                        score, player = self.minimaxAB(depth - 1, False,
    alpha, beta, board) # Point of entry into the recursion
106                        if score >= best_score:
107                            best_score = score
108                            best_move = leaf_Node
109                    move_ai.undo_capture(board, leaf_Node[0], leaf_Node[1],
    leaf_Node[2], leaf_Node[3], leaf_Node[4])
110                # Considering non-capture move.
111                else:
112                    move_ai.move(board, leaf_Node[0], leaf_Node[1],
    leaf_Node[2], leaf_Node[3])
113
114                    score, player = self.minimaxAB(depth - 1, False, alpha,
    beta, board) # Point of entry into the recursion
115                    if score >= best_score:
116                        best_score = score
117                        best_move = leaf_Node
118
119                    move_ai.undo_move(board, leaf_Node[0], leaf_Node[1],
    leaf_Node[2], leaf_Node[3])
120                # MAX updates alpha if current value is larger than alpha.
121                alpha = max(alpha, score)
122                if alpha >= beta:
123                    break # beta cut-off
124            return score, best_move
125
126        # Considering the min-Player.
127        else:
128            best_score = math.inf
129            score = math.inf
130            player, capture_move = self.player.calc_player(board)
131
132            for leaf_Node in player:
133                # Considering capture move.
134                if capture_move is True:
135                    move_ai.capture(board, leaf_Node[0], leaf_Node[1],
    leaf_Node[2], leaf_Node[3], leaf_Node[4])
136                    # Considering capture move, especially, 'multi-leg moves
    '.
137                    can_more_capture_move = board[leaf_Node[0]][leaf_Node
    [1]].can_token_capture(board)
138                    if can_more_capture_move:
139                        score, player = self.minimaxAB(depth - 1, False,
    alpha, beta, board) # Point of entry into the recursion
```

```
                        if score < best_score:
                            best_score = score
                            best_move = leaf_Node
                    # only once capture move
                    else:
                        score, player = self.minimaxAB(depth - 1, True, alpha, beta, board) # Point of entry into the recursion
                        if score <= best_score:
                            best_score = score
                            best_move = leaf_Node

                    move_ai.undo_capture(board, leaf_Node[0], leaf_Node[1], leaf_Node[2], leaf_Node[3], leaf_Node[4])
                # Considering non-capture move.
                else:
                    move_ai.move(board, leaf_Node[0], leaf_Node[1], leaf_Node[2], leaf_Node[3])
                    score, player = self.minimaxAB(depth - 1, True, alpha, beta, board) # Point of entry into the recursion
                    if score <= best_score:
                        best_score = score
                        best_move = leaf_Node
                    move_ai.undo_move(board, leaf_Node[0], leaf_Node[1], leaf_Node[2], leaf_Node[3])
                # MIN updates beta if current value is less than beta.
                beta = min(beta, score)
                if alpha >= beta:
                    break # alpha cut-off

            return score, best_move
```

Listing 10: Search.py

A.11. Token.py

```
# Token.py
from Constants import *



class Token:
    # Class for handle the general value for token.
    def __init__(self, posy, posx, color):
        self.posy = posy
        self.posx = posx
        self.color = color
        # if the token's color is white(used by AI player), since white token's position is top side of the board, direction becomes -1 (go down)
        if self.color == WHITE:
            self.direction = -1 # row - 1
        # if the token's color is black(used by human player), since white token's position is down side of the board, direction becomes +1 (go up)
        else:
            self.direction = 1 # row + 1

    # return the color value of the token.
    def calc_color(self):
        return self.color
```

```
22
23      # Calculate the possible moves of tokens on given tile through iterate
    sublists of the board.
24      def calc_possible_moves(self, board):
25          # store the each position of possible moves.
26          possible_moves = []
27          for tile_x in range(len(board)):
28              column = []
29              for tile_y in range(len(board)):
30                  proper_move = board[self.posy][self.posx].is_possible_move(
    board, tile_x, tile_y)
31                  column.append(proper_move)
32              possible_moves.append(column)
33          return possible_moves
34
35      # Calculate the possible capture moves of tokens on given tile through
    iterate sublists of the board.
36      def calc_possible_captures(self, board):
37          # store the each position of possible capture moves.
38          possible_captures = []
39          for tile_x in range(len(board)):
40              column = []
41              for tile_y in range(len(board)):
42                  proper_capture = board[self.posy][self.posx].
    is_capture_available(board, tile_x, tile_y)
43                  column.append(proper_capture)
44              possible_captures.append(column)
45          return possible_captures
46
47      # Calculate the all possible moves of tokens on given tile through
    iterate sublists of the board.
48      # Return the boolean value that is indicate the capturing move is
    possible or not.
49      def calc_all_possible_moves(self, board):
50          # Get the possible captures move.
51          possible_captures = self.calc_possible_captures(board)
52          if any(True in sublist for sublist in possible_captures):
53              capture = True
54              return possible_captures, capture
55          # Get the possible move.
56          else:
57              possible_moves = self.calc_possible_moves(board)
58              capture = False
59              return possible_moves, capture
60
61      # Check the whether the move is valid or not.
62      def is_possible_move(self, board, tile_y, tile_x):
63          if isinstance(board[tile_y][tile_x], Token):
64              return False
65          if self.posy - tile_y == self.direction and abs(self.posx - tile_x)
    == 1:
66              return True
67          else:
68              return False
69
70      # Update the state of board data structure and the token's updated
    position.
71      def make_move(self, board, tile_y, tile_x):
72          board[self.posy][self.posx], board[tile_y][tile_x] = board[tile_y][
```

```
    tile_x], board[self.posy][self.posx]
73          self.posy = tile_y
74          self.posx = tile_x
75
76      def can_token_capture(self, board):
77          possible_captures = []
78          for tile_x in range(len(board)):
79              column = []
80              for tile_y in range(len(board)):
81                  proper_capture = board[self.posy][self.posx].
    is_capture_available(board, tile_x, tile_y)
82                  column.append(proper_capture)
83              possible_captures.append(column)
84
85          if any(True in sublist for sublist in possible_captures):
86              return True
87          else:
88              return False
```

Listing 11: Token.py

5. References

References

[1] Checkers Rules and Layout [online] Available at: https://a4games.company/checkers-rules-and-layout/ [Online; Accessed 10-Nov-2021]

[2] How to install Pygame [online] Available at: https://www.geeksforgeeks.org/how-to-install-pygame-in-windows/ [Online; Accessed 10-Nov-2021]

[3] Python/Pygame Checkers Tutorial (Part 1) - Drawing the Board [online] Available at: https://www.youtube.com/watch?v=vnd3RfeG3NM&t=1132s [Online; Accessed 13-Nov-2021]

[4] Pygame documentation [online] Available at: https://www.pygame.org/docs/ref/surface.html [Online; Accessed 13-Nov-2021]

[5] PyGame: A Primer on Game Programming in Python [online] Available at: https://realpython.com/pygame-a-primer/ [Online; Accessed 13-Nov-2021]

[6] Minimax Algorithm in Game Theory [online] Available at: https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-1-introduction/ [Online; Accessed 15-Nov-2021]

[7] Game theory — Minimax [online] Available at: https://towardsdatascience.com/game-theory-minimax-f84ee6e4ae6e [Online; Accessed 15-Nov-2021]

[8] Minimax with Alpha-Beta Pruning in Python [online] Available at: https://stackabuse.com/minimax-and-alpha-beta-pruning-in-python/ [Online; Accessed 15-Nov-2021]

[9] How to create a text input box with pygame [online] Available at: https://www.geeksforgeeks.org/how-to-create-a-text-input-box-with-pygame/ [Online; Accessed 20-Nov-2021]

[10] PyGame Beginner Tutorial in Python - Adding Buttons [online] Available at: https://www.youtube.com/watch?v=G8MYGDf_9ho [Online; Accessed 20-Nov-2021]

[11] Download free sound effect [online] Available at: https://mixkit.co/free-sound-effects [Online; Accessed 20-Nov-2021]

[12] Python Pygame: Adding Background Music to a Python Game https://www.askpython.com/python-modules/pygame-adding-background-music [Online; Accessed 20-Nov-2021]