OTHELLO DEVELOPMENT

PRATICAL ASSIGMENT

Victor Demessance | RTU – Artificial Intelligence | April 2023

Link: https://github.com/Victordmss/Othello_minimaxAI

Contents

PRESENTATION

PROJECT:

The goal of this project is to develop a game and implement an intelligent bot that allows the user to play alone.

CONSTRAINTS:

There are several constraints. The aim of these is to directly apply the course to the project and not to take the wrong direction. **Moreover, the game developed is intended to be used by all types of users and must therefore be made up of a good UX/UI**.

In first, the software must provide the following possibilities for the user:

- To choose who starts the game: human or computer.
- To start a new game after the completion of the previous game.

Moreover, the student must implement the following in the software:

- Generation of a state space graph of the game.
- Heuristic evaluation function *(because we will deal with a partly available state space graph of the game at each turn)*
- A game algorithm considered in the course *(Minimax in our case)*.

OTHELLO:

The game I chose is called **Othello**. It is a **two-person zero-sum game** very famous in France. This game isn't so easy to program because it needs a process that can find and flip the necessary coins.

SOLUTION

ARCHITECTURE:

The solution is developed in Python. The project is made of several files for each different class allowing the game to work (buttons, grid...). The main data structure used is a list representing the matrix of the game (in other words the grid with the colored pieces).

The different methods and functions of each class are explained in the comments of the program (code below).

For the Othello game class, different structures are used.

To register the players, we use a dictionary (keeping several tuples according to the following convention (key, color, AI). This convention allows us to know the identification key of the piece on the game grid, the color of this one and the control of this piece by the AI or not). This dictionary allows to reduce the number of attributes of the class by merging the different players in the same attribute.

The game states allows to give dynamic information about the current state of the game at any time. Moreover, when used with the "update_background()" method, this attribute allows to directly change the background with the file corresponding to the game state.

As for the players and for the same reasons of optimization, the buttons of the graphical interface of the game are kept in a dictionary. These buttons are built according to the initialization convention of the Button class.

Finally, the game board is built thanks to the Board class, which defines the architecture and the backend of the grid that will be graphically displayed on the game window.

HEURISTIC EVALUATION:

The heuristic evaluation is based on several pieces of information that I found on the internet and in books.

First, the most important thing is the number of pieces flipped by the move. In fact, for each move a player makes, at least one piece must be flipped. This way, each flipped piece represents one more point in the player's score and one less in the opponents. This value represents the most important weight of the movement chosen by the AI.

Moreover, we calculate the potential of each move thanks to a weighted matrix of the positions on the grid. This matrix is defined thanks to data found on the internet, in several projects of construction of heuristic function of evaluation in the context of the game of Othello. This weighted matrix is constant and can be found on the board.py file.

MINIMAX ALGORITHM:

The minimax algorithm used in my project is simple. The goal is to find the maximin value, that is “the highest value that the player can be sure to get without knowing the actions of the other players; equivalently, it is the lowest value the other players can force the player to receive when they know the player's action.” (Wikipedia, *Minimax*). That algorithm has been learnt in course. Consequently, I am not going to present it in this report. Its operation is quite classic and does not require further explanation.

In our case, to be able to calculate all the possible move in a good time, we will not go through more than 3 levels in the constructed tree. For the easy level of the AI, we will work with just 1 level, for the medium 2 and for the hard level 3 levels.

INTERFACE:

The game interface has been built on python with the help of the Pygame module. This module is a library for the creation of video games. It works in a similar way to Tkinter, but this library is more focused on drawing and updating frames to simulate the movement and evolution of the game.

In this way, the program displays a background matching the state of the game and updates at each time step the logical grid state managed by the board object of the Board class defined in the board.py file. This display evolves by linking the identification keys of the players (filling the matrix) to a dynamic circle of their color on the board.

Finally, many buttons and displays work in a way that allows the player a more pleasant and dynamic game experience, improving the gameplay and usability of the game developed.

CODE:

Othello.py:

```
import pygame
from button import Button
from board import Board
import webbrowser
import sys

RULES_URL = "https://www.worldothello.org/about/about-othello/othello-rules/official-rules/english"

SCREEN_SIZE = (1000, 600)

GAME_STATES = {"LAUNCHING": 'launching',
               "PLAYING": 'playing',
               "ENDING": 'ending',
               "P1WON": "p1won",
               "P2WON": "p2won",
               "DRAW": "draw",
               "REVIEW": "review", }


class Othello:  # Class representing the functioning of the game of Othello
    def __init__(self, row_count, column_count):
        # Game parameters
        # Players
        self.players = {
            "white_player": {"key": 1, "color": (255, 255, 255), "AI": False},
            "black_player": {"key": 2, "color": (0, 0, 0), "AI": True},
        }
        # Game state
        self.current_player = self.players["black_player"]["key"]  # Black player always starts
        self.game_state = GAME_STATES["LAUNCHING"]
        self.ai_start = False
        self.difficulty = 1   # Default difficulty is easy (1 = easy, 2 = medium, 3 = hard)

        # Buttons
        self.buttons = {
            # Launching buttons
            "black_button" : Button(184, 245, 352, 286),
            "white_button" : Button(714, 248, 357, 820),
            # Playing buttons
            "reset_button" : Button(14, 456, 518, 196),
            "rules_button" : Button(790, 457, 516, 973),
            # Ending buttons
            "review_button" : Button(14, 528, 586, 196),
            # Difficulty button
            "easy_diff_button" : Button(14, 387, 445, 69),
            "medium_diff_button" : Button(77, 387, 445, 133),
            "hard_diff_button" : Button(140, 387, 445, 196),
        }

        # Creating board
        self.row_count = row_count
        self.column_count = column_count
        self.board = Board(self.row_count, self.column_count, (89, 139, 44), SCREEN_SIZE)

        # Initialisation of the game
        # Creation of the screen
        pygame.init()
        screen = pygame.display.set_mode(SCREEN_SIZE)
        self.font = pygame.font.Font(pygame.font.get_default_font(), 40)
        pygame.display.set_caption('Abalone')
        # Launching of the game
        self.update_background(screen)
        self.launching_othello(screen)

    # ------------------------------------------------------------------------------------------------------------------
    # Gaming process methods

    # This method launches Othello and lets the user choose who starts
    def launching_othello(self, screen: pygame.Surface):
        while True:
            for event in pygame.event.get():
                if event.type == pygame.QUIT:  # Quit the game
                    pygame.quit()
                    sys.exit()
                if event.type == pygame.MOUSEBUTTONDOWN:  # Click are used to start the game and choose his color
                    mouse_position = event.pos
                    if self.buttons["black_button"].is_clicked(mouse_position):  # User chooses black, he starts
                        self.players["white_player"]["AI"] = True
                        self.ai_start = False
                        self.players["black_player"]["AI"] = False
                    if self.buttons["white_button"].is_clicked(mouse_position):  # User chooses white, AI starts
                        self.players["white_player"]["AI"] = False
                        self.ai_start = True
                        self.players["black_player"]["AI"] = True
                    self.game_state = GAME_STATES["PLAYING"]
```

```
                    self.playing_othello(screen)  # Now start Othello

    # This method represents the main process of the Othello game
    def playing_othello(self, screen: pygame.Surface):
        self.init_game_background(screen)
        self.display_available_move(screen, self.players["white_player"]["key"])
        if self.ai_start:  # In case the user has chosen white, let's play with AI first
            self.board.is_there_valid_move(self.players["black_player"]["key"], self.players["white_player"]["key"])
            self.AI_turn(screen)


        while True:  # Global loop for the game's process
            for event in pygame.event.get():
                if event.type == pygame.QUIT:  # Quit the game
                    pygame.quit()
                    sys.exit()
                if event.type == pygame.MOUSEBUTTONDOWN:  # Click are used to play and place your coin
                    mouse_position = event.pos  # Click position
                    #print(mouse_position)
                    if self.buttons["rules_button"].is_clicked(mouse_position):  # Let's show the rules of the game
                        webbrowser.open(RULES_URL)  # Let's open the official website of Othello's rules
                    elif self.buttons["reset_button"].is_clicked(mouse_position):  # If user decides to reset the game
                        self.reset_game(screen)
                    elif self.buttons["easy_diff_button"].is_clicked(mouse_position):  # Difficulty changes to easy
                        self.difficulty = 1
                    elif self.buttons["medium_diff_button"].is_clicked(mouse_position):  # Difficulty changes to medium
                        self.difficulty = 2
                    elif self.buttons["hard_diff_button"].is_clicked(mouse_position):  # Difficulty changes to hard
                        self.difficulty = 3
                    elif self.board.is_clicked(mouse_position):  # Check If the click is inside the board
                        if self.play_user(mouse_position, screen):  # Play if the move is allowed
                            self.display_available_move(screen, self.current_player)
                            self.next_turn(screen)
                            self.show_score(screen)
                            self.is_game_ended(screen)
                            pygame.display.update()  # Updating screen display because AI has to play
                            self.AI_turn(screen)    # AI turn process
                            self.is_game_ended(screen)
                pygame.display.update()

    # This method is the main user turn process. We manage a verification process and the modification of the board
    def play_user(self, mouse_position: (float, float), screen: pygame.Surface):
        column_click, row_click = self.convert_click_to_position(mouse_position)  # We convert the click in a position
        if (row_click, column_click) in self.board.available_moves:  # If the move is available
            self.board.grid[row_click][column_click] = self.current_player
            self.board.update_grid(row_click, column_click, self.current_player)
            self.update_board_display(screen)
            return True  # Move done with success
        return False  # Move invalid

    # This method ends the game, calculates who won and brings to the ending page
    def ending_othello(self, screen: pygame.Surface):
        white_player_score = self.board.count_points(self.players["white_player"]["key"])
        black_player_score = self.board.count_points(self.players["black_player"]["key"])
        if white_player_score > black_player_score:  # Calculate if white player won
            self.game_state = GAME_STATES["P1WON"]
        elif white_player_score < black_player_score:  # Calculate if black player won
            self.game_state = GAME_STATES["P2WON"]
        else:  # Calculate if nobody won because of a draw
            self.game_state = GAME_STATES["DRAW"]
        self.update_background(screen)  # Updating to the good ending page depending on the result of the game
        while True:  # Ending game process, allowing the user to review or to reset the game
            for event in pygame.event.get():
                if event.type == pygame.QUIT:  # Quit the game
                    pygame.quit()
                    sys.exit()
                if event.type == pygame.MOUSEBUTTONDOWN:  # Click are used to play and place your coin
                    mouse_position = event.pos
                    if self.buttons["reset_button"].is_clicked(mouse_position):  # Reset the game
                        self.reset_game(screen)
                    elif self.buttons["review_button"].is_clicked(mouse_position):  # Review the game
                        self.review_game(screen)

    # ------------------------------------------------------------------------------------------------------------------
    # AI implementation methods

    # This method manages the AI turn process
    def AI_turn(self, screen):
        pygame.time.wait(500)
        self.play_AI()  # AI playing process, using minimax algorithm
        self.display_available_move(screen, self.current_player)
        self.update_board_display(screen)
        self.next_turn(screen)
        self.show_score(screen)
        pygame.display.update()

    # This method is the main AI turn process. We manage the selection of the move and the modification of the board
    def play_AI(self):
        if self.current_player == self.players["white_player"]["key"]:
            other_player = self.players["black_player"]["key"]
        else:
            other_player = self.players["white_player"]["key"]
```

```
        best_move = [-1, -1]  # Default best move, allow us to know if the minimax algorithm doesn't work
        max_point = float('-inf')  # Represent the points of the best found move
        for move in self.board.available_moves:
            temp_board = self.board.copy_board(SCREEN_SIZE)  # We copy the board because we want to simulate moves
            temp_board.grid[move[0]][move[1]] = self.current_player  # Simulation of one of the available move
            flipped_coin = temp_board.update_grid(move[0], move[1], self.current_player)  # How many coin did it
flipped
            temp_board.is_there_valid_move(other_player, self.current_player)  # Updating of the new available moves
            move_points = self.minimax(temp_board, other_player, False, flipped_coin, depth=self.difficulty)
            if move_points > max_point:
                max_point = move_points  # Then this move is currently the best that we found, we save it
                best_move = move
        if best_move != [-1, -1]:  # If we have found a move
            self.board.grid[best_move[0]][best_move[1]] = self.current_player  # Modification of the board
            self.board.update_grid(best_move[0], best_move[1], self.current_player)  # Updating of the board
            return True
        return False  # Not any move has been found

    # Implementation of the minimax algorithm for Othello game
    def minimax(self, board: Board, player: int, maximizing_player: int, turned_coin: int, depth: int):
        if player == self.players["white_player"]["key"]:
            other_player = self.players["black_player"]["key"]
        else:
            other_player = self.players["white_player"]["key"]

        if depth == 0 or board.available_moves == []:  # If we have a leaf node
            total = self.evaluate_board(board, player, turned_coin)  # We evaluate the board
            return total

        if maximizing_player:  # If we are with the maximizing player
            best_value = float('-inf')  # We want to find the best board evaluation
            for move in board.available_moves:
                temp_board = board.copy_board(SCREEN_SIZE)  # We copy the board because we want to simulate moves
                temp_board.grid[move[0]][move[1]] = player  # Simulation of one of the available move
                turned_coin = temp_board.update_grid(move[0], move[1], player)  # How many coin did it flipped
                temp_board.is_there_valid_move(other_player, player)  # Updating of the new available moves
                value = self.minimax(temp_board, other_player, False, turned_coin, depth - 1)  # Minimax process
                best_value = max(best_value, value)  # Value of this node is the best value that we found
        else:  # If we are with the minimizing player
            best_value = float('+inf')  # We want to find the worth board evaluation
            for move in board.available_moves:
                temp_board = board.copy_board(SCREEN_SIZE)  # We copy the board because we want to simulate moves
                temp_board.grid[move[0]][move[1]] = player  # Simulation of one of the available move
                turned_coin = temp_board.update_grid(move[0], move[1], player)  # How many coin did it flipped
                temp_board.is_there_valid_move(other_player, player)  # Updating of the new available moves
                value = self.minimax(temp_board, other_player, True, turned_coin, depth - 1)  # Minimax process
                best_value = min(best_value, value)  # Value of this node is the worth value that we found
        return best_value  # We return the value for each possible move of the grid


    # ------------------------------------------------------------------------------------------------------------------
-
    # Control and interruption methods

    # This method tests if the game is ended
    def is_game_ended(self, screen):
        if not self.board.available_moves:  # If there are no available move anymore, game ends
            self.game_state = GAME_STATES["ENDING"]
            self.ending_othello(screen)

    # This method resets the game and brings back to the launching page
    def reset_game(self, screen: pygame.Surface):
        self.current_player = self.players["black_player"]["key"]  # White always starts
        self.game_state = GAME_STATES["LAUNCHING"]
        self.update_background(screen)
        self.board = Board(self.row_count, self.column_count, (89, 139, 44), SCREEN_SIZE)  # New board
        self.launching_othello(screen)

    # This method allows the user to review the previous game, and then to reset it
    def review_game(self, screen: pygame.Surface):
        winner = self.game_state
        self.game_state = GAME_STATES["REVIEW"]
        self.update_background(screen)
        self.display_user_color(screen)
        self.draw_grid(screen)
        self.update_board_display(screen)
        self.show_score(screen)
        if winner == 'p1won':  # Show who won
            pygame.draw.circle(screen, self.players["white_player"]["color"],
                               (105,
                                117),
                               self.board.radius * 1.25)
        elif 'p2won':  # Show who won
            pygame.draw.circle(screen, self.players["black_player"]["color"],
                               (105,
                                117),
                               self.board.radius * 1.25)
        pygame.display.update()

    # This method manage the end of a turn and switch the current player
    def next_turn(self, screen: pygame.Surface):
        if self.current_player == self.players["white_player"]["key"]:
            self.current_player = self.players["black_player"]["key"]
```

```
        else:
            self.current_player = self.players["white_player"]["key"]
        self.change_player_indicator(screen)  # We change the player indicator that it displays on the screen

    # ------------------------------------------------------------------------------------------------------------------
    # Display methods

    # This method is called at the beginning of the playing part of the game in order to create the displayed board
    def init_game_background(self, screen: pygame.Surface):
        self.update_background(screen)  # Updating the background
        self.display_user_color(screen)  # Show the user main color to remind it to him
        self.board.grid[3][3] = self.players["white_player"]["key"]  # Place the starting pieces
        self.board.grid[4][4] = self.players["white_player"]["key"]  # Place the starting pieces
        self.board.grid[4][3] = self.players["black_player"]["key"]  # Place the starting pieces
        self.board.grid[3][4] = self.players["black_player"]["key"]  # Place the starting pieces
        self.draw_grid(screen)  # Draw the grid on the screen
        # This section has the same goal as self.update_board_display. However, we don't have to analyse all the grid
        self.draw_circle(3, 3, screen, self.players["white_player"]["color"], self.board.radius)
        self.draw_circle(4, 4, screen, self.players["white_player"]["color"], self.board.radius)
        self.draw_circle(4, 3, screen, self.players["black_player"]["color"], self.board.radius)
        self.draw_circle(3, 4, screen, self.players["black_player"]["color"], self.board.radius)
        self.change_player_indicator(screen)  # Let's build the player display indicator
        pygame.display.update()

    # This method changes the player indicator circle that it display on the top right of the screen
    def change_player_indicator(self, screen: pygame.Surface):
        if self.current_player == self.players["white_player"]["key"]:  # If the current player is the white player
            pygame.draw.circle(screen, self.players["white_player"]["color"],
                               (105,
                                117),
                               self.board.radius * 1.25)
        else:  # If the current player is the black player
            pygame.draw.circle(screen, self.players["black_player"]["color"],
                               (105,
                                117),
                               self.board.radius * 1.25)

    # This method displays the user color on the right part of the board in order to remind it to him during the game
    def display_user_color(self, screen: pygame.Surface):
        if not self.players["white_player"]['AI']:  # If the player is the white player
            pygame.draw.circle(screen, self.players["white_player"]["color"],
                               (105,
                                210),
                               self.board.radius * 1.25)
        else:  # If the player is the black player
            pygame.draw.circle(screen, self.players["black_player"]["color"],
                               (105,
                                210),
                               self.board.radius * 1.25)

    # This method draws the green grid on the board with a sequence of green rectangle
    def draw_grid(self, screen: pygame.Surface):
        for column in range(self.column_count):
            for row in range(self.row_count):
                pygame.draw.rect(screen, self.board.color,
                                 (self.board.left_board_side + column * self.board.box_size * 1.03,
                                  self.board.top_board_side + self.board.box_size * row * 1.03,
                                  self.board.box_size,
                                  self.board.box_size))

    # This method shows all the available moves of the board with little grey circles
    def display_available_move(self, screen: pygame.Surface, last_player_key: int):
        # Generating other_coin_key
        if last_player_key == self.players["black_player"]["key"]:
            next_player_key = self.players["white_player"]["key"]
        else:
            next_player_key = self.players["black_player"]["key"]

        # Deleting older display
        self.reset_available_moves(screen)

        # Testing all the bord
        self.board.is_there_valid_move(next_player_key, last_player_key)

        # Drawing available moves
        for (row, col) in self.board.available_moves:
            self.draw_circle(col, row, screen, (200, 200, 200), self.board.radius // 2)

    # This method deletes all the little circles of available moves that we created before
    def reset_available_moves(self, screen: pygame.Surface):
        for row, col in self.board.available_moves:
            if self.board.grid[row][col] == 0:
                self.draw_circle(col, row, screen, self.board.color, self.board.radius)
        self.board.available_moves = []  # Then, we delete all the current available moves

    # This method update the board while creating circle of the good color
    def update_board_display(self, screen: pygame.Surface):
        for row in range(self.row_count):
            for col in range(self.column_count):
                if self.board.grid[row][col] == self.players["white_player"]["key"]:
                    self.draw_circle(col, row, screen, self.players["white_player"]["color"], self.board.radius)
                elif self.board.grid[row][col] == self.players["black_player"]["key"]:
```

```
                    self.draw_circle(col, row, screen, self.players["black_player"]["color"], self.board.radius)

    # This method shows the score of the game on the left part of the screen
    def show_score(self, screen):
        pygame.draw.rect(screen, (198, 184, 168), (48, 285, 50, 50))    # Use to erase last score display
        pygame.draw.rect(screen, (198, 184, 168), (125, 285, 50, 50))   # Use to erase last score display
        white_score = self.font.render(
            str(self.board.count_points(self.players["white_player"]["key"])),
            True,
            (255, 255, 255)
        )
        black_score = self.font.render(
            str(self.board.count_points(self.players["black_player"]["key"])),
            True,
            (0, 0, 0)
        )
        screen.blit(white_score, (50, 285))
        screen.blit(black_score, (125, 285))

    # This method updates game's background using the global state of the game
    def update_background(self, screen: pygame.Surface):
        fond = pygame.image.load(f'assets/{self.game_state}_background.png')
        fond = fond.convert()
        screen.blit(fond, (0, 0))
        pygame.display.flip()
        pygame.display.update()

    # ---------------------------------------------------------------------------------------------------------------------
-
    # Tool methods

    # This method is a shortcut to draw a circle at a specific (row, column) quickly
    def draw_circle(self, col: int, row: int, screen: pygame.Surface, color: (int, int, int), radius: int):
        pygame.draw.circle(screen, color,
                           (self.board.left_board_side + col * self.board.box_size * 1.03 + self.board.box_size // 2,
                            self.board.top_board_side + self.board.box_size * row * 1.03 + self.board.box_size // 2),
                           radius)

    # This method convert a click in a tuple integer that represent the position of the click in the grid
    def convert_click_to_position(self, mouse_position: (float, float)):
        column_click = (mouse_position[0] - self.board.left_board_side) / self.board.box_size / 1.03
        column_click = self.round_click_pos(column_click)
        row_click = (mouse_position[1] - self.board.top_board_side) / self.board.box_size / 1.03
        row_click = self.round_click_pos(row_click)
        return column_click, row_click

    def round_click_pos(self, x):  # This function convert a float into an integer by truncating at the unit
        if x < 1:
            x = 0
        elif x < 2:
            x = 1
        elif x < 3:
            x = 2
        elif x < 4:
            x = 3
        elif x < 5:
            x = 4
        elif x < 6:
            x = 5
        elif x < 7:
            x = 6
        else:
            x = 7
        return x

    # This method is a heuristic evaluation method that allow us to put a score for a specific grid
    def evaluate_board(self, board, player, turned_coin):
        total = 0
        for row in range(board.row_count):
            for col in range(board.column_count):
                if board.grid[row][col] == player:
                    total += board.grid[row][col] * board.square_weight[row][col]
        return total + 1.5 * turned_coin
```

board.py:

```
import numpy as np

# Global constant variable

DIRECTIONS = [  # Represents all the possible directions in a two-dimensional space
    [0, 1],
    [0, -1],
    [1, 0],
    [-1, 0],
    [1, 1],
    [-1, -1],
    [-1, 1],
    [1, -1],
]

SQUARE_WEIGHTS = [  # Matrix representing the game grid weighted by the importance of the positions to win
    [120, -20, 20, 5, 5, 20, -20, 120],
    [-20, -40, -5, -5, -5, -5, -40, -20],
    [20, -5, 15, 3, 3, 15, -5, 20],
    [5, -5, 3, 3, 3, 3, -5, 5],
    [5, -5, 3, 3, 3, 3, -5, 5],
    [20, -5, 15, 3, 3, 15, -5, 20],
    [-20, -40, -5, -5, -5, -5, -40, -20],
    [120, -20, 20, 5, 5, 20, -20, 120],
]


# This class represents the game board, composed of the logical grid and the board display parameters.
class Board:
    def __init__(self, row_count, column_count, color, screen_size):
        # Logical board parameters
        self.row_count = row_count    # Number of rows of the board
        self.column_count = column_count    # Number of columns of the board
        self.grid = np.zeros((self.row_count, self.column_count))   # Logical grid of the board
        self.available_moves = []   # List of all the possible moves for the current player at each moment of the game
        self.square_weight = SQUARE_WEIGHTS

        # Display board parameters
        self.color = color   # Color of the board
        self.box_size = 50   # Size of each box of the board (they are squares, so we just need one value)
        self.radius = int(self.box_size / 2 - 5)    # Radius of the board pieces (no matter the color/player)
        self.board_size = (self.row_count * self.box_size, self.column_count * self.box_size)   # Size of the board
        # Screen is (1000*600)
        self.left_board_side = screen_size[0]//2 - (self.box_size * self.column_count) // 2   # Left board display side
        self.top_board_side = screen_size[1]//2 - (self.box_size * self.row_count) // 2   # Top board display side
        self.bottom_board_side = screen_size[1]//2 + (self.box_size * self.row_count) // 2   # Bottom board display
side
        self.right_board_side = screen_size[0]//2 + (self.box_size * self.column_count) / 2   # Right board display
side

    # ------------------------------------------------------------------------------------------------------------------
-
    # Security methods

    # This method just return a boolean value that indicates if the position is inside the grid or not
    def overflow(self, x: int, y: int):
        return not (0 <= x <= self.column_count - 1 and 0 <= y <= self.row_count - 1)

    # This method just return a boolean value that indicates if the click is inside the display board
    def is_clicked(self, mouse_position: (float, float)):
        return self.top_board_side <= mouse_position[1] <= self.bottom_board_side and \
            self.left_board_side <= mouse_position[0] <= self.right_board_side

    # This method allows to copy the board without memory reference
    def copy_board(self, screen_size):
        board = Board(self.row_count, self.column_count, (89, 139, 44), screen_size)
        board.grid = np.copy(self.grid)
        return board

    # ------------------------------------------------------------------------------------------------------------------
-
    # Processing grid methods

    # This method allows us to determine the moves that are available in the next turn
    # and to save them in the attribute of the class (available_move : array)
    def is_there_valid_move(self, next_player_key: int, last_player_key: int):
        self.available_moves = []   # Reset the available moves from the previous turn
        for row in range(self.row_count):
            for col in range(self.column_count):
                if self.grid[row][col] == next_player_key:  # if a box is filled by the next player
                    for x_direction, y_direction in DIRECTIONS:    # Let's test in all the directions
                        x = row + x_direction
                        y = col + y_direction
                        if not self.overflow(x, y) and self.grid[x][y] == last_player_key:
                            x += x_direction
                            y += y_direction
                            while not self.overflow(x, y) and self.grid[x][y] == last_player_key:
                                x += x_direction
                                y += y_direction
```

```
                            if not self.overflow(x, y) and self.grid[x][y] == 0 and (x, y) not in self.available_moves:
                                # If this position allows the next player to convert the opponent's pieces
                                # and the position is on the grid, then it is a valid move. We save it
                                self.available_moves.append((x, y))

    # This method allows us to count the points of a specific player whose key we have passed in parameter
    def count_points(self, piece_key: int):
        s = 0  # s represents the amount of piece of his color that the player has on the board
        for row in range(self.row_count):
            for col in range(self.column_count):
                if self.grid[row][col] == piece_key:
                    s += 1
        return s

    # ------------------------------------------------------------------------------------------------------------------
    # Grid editing methods

    # After each move, this method is called to update the grid by flipping the necessary pieces
    # This function also return the number of pieces that have been flipped
    def update_grid(self, row_click: int, column_click: int, player_key: int):
        s = 0   # Represent the number of pieces that will be flipped
        if player_key == 1:
            other_key = 2
        else:
            other_key = 1
        for x_direction, y_direction in DIRECTIONS:  # Let's test in all the directions
            x = row_click + x_direction
            y = column_click + y_direction
            if not self.overflow(x, y) and self.grid[x][y] == other_key:
                x += x_direction
                y += y_direction
                while not self.overflow(x, y) and self.grid[x][y] == other_key:
                    x += x_direction
                    y += y_direction
                if not self.overflow(x, y) and self.grid[x][y] == player_key:
                    # If we find another piece of the same color after passing through pieces of a different colour
                    # Then we do the same path in backwards while flipping the pieces we find
                    # until we come back to the starting position
                    while x != row_click or y != column_click:
                        x -= x_direction
                        y -= y_direction
                        s += 1
                        self.grid[x][y] = player_key
        return s    # We return the number of pieces that we flipped

    # This method resets the logical grid of the board
    def reset_board(self):
        self.grid = np.zeros((self.row_count, self.column_count))
```

button.py:

```
# This class represents the buttons available in the game
class Button:
    def __init__(self, left_side, top_side, bottom_side, right_side):
        self.left_side = left_side  # Left border side of the button
        self.top_side = top_side    # Top border side of the button
        self.bottom_side = bottom_side  # Bottom border side of the button
        self.right_side = right_side    # Right border side of the button

    # This method just return a boolean value that indicates if the click is inside the button
    def is_clicked(self, mouse_position):
        return self.left_side <= mouse_position[0] <= self.right_side and \
            self.top_side <= mouse_position[1] <= self.bottom_side
```

Main.py:

```
from othello import Othello     # importing the Game Class from the file

ROW_COUNT = 8
COLUMN_COUNT = 8

othello = Othello(ROW_COUNT, COLUMN_COUNT)
```

USER MANUAL

RULES:

The rules of Othello are **simple**, here is a summary:

The object of the game is to have the majority of discs of your own color at the end of the game.

A move consists of "outflanking" your opponent's disc(s), then flipping the outflanked disc(s)to your color. To outflank means to place a disc on the board so that your opponent's row (or rows) of disc(s) is bordered at each end by a disc of your color. (A "row" may be made up of one or more discs).

Here's one example:

White disc A was already in place on the board. The placement of white disc B outflanks the row of three black discs. White flips the outflanked discs and now the row looks like this:

<u>Now, let's see the rules for playing Othello:</u>

1. Black always moves first.

2. If on your turn you cannot outflank and flip at least one opposing disc, the game is ended. However, if a move is available to you, you may not forfeit your turn.

4. Players may not skip over their own color disc(s) to outflank an opposing disc. (Figure 4).

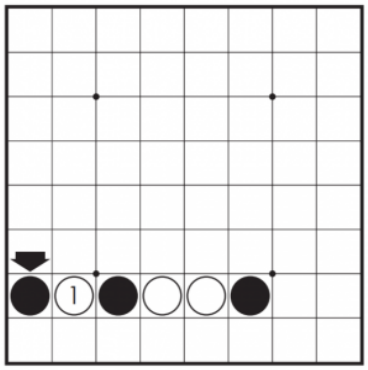


Figure 4: This disc outflanks and flips White disc 1 ONLY.

5. Disc(s) may only be outflanked as a direct result of a move and must fall in the direct line of the disc placed down. *(See Figures 5 and 6).*

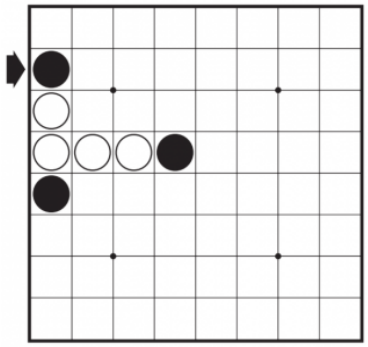


Figure 5: Disc placed here.

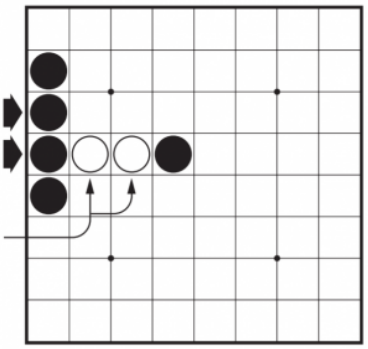


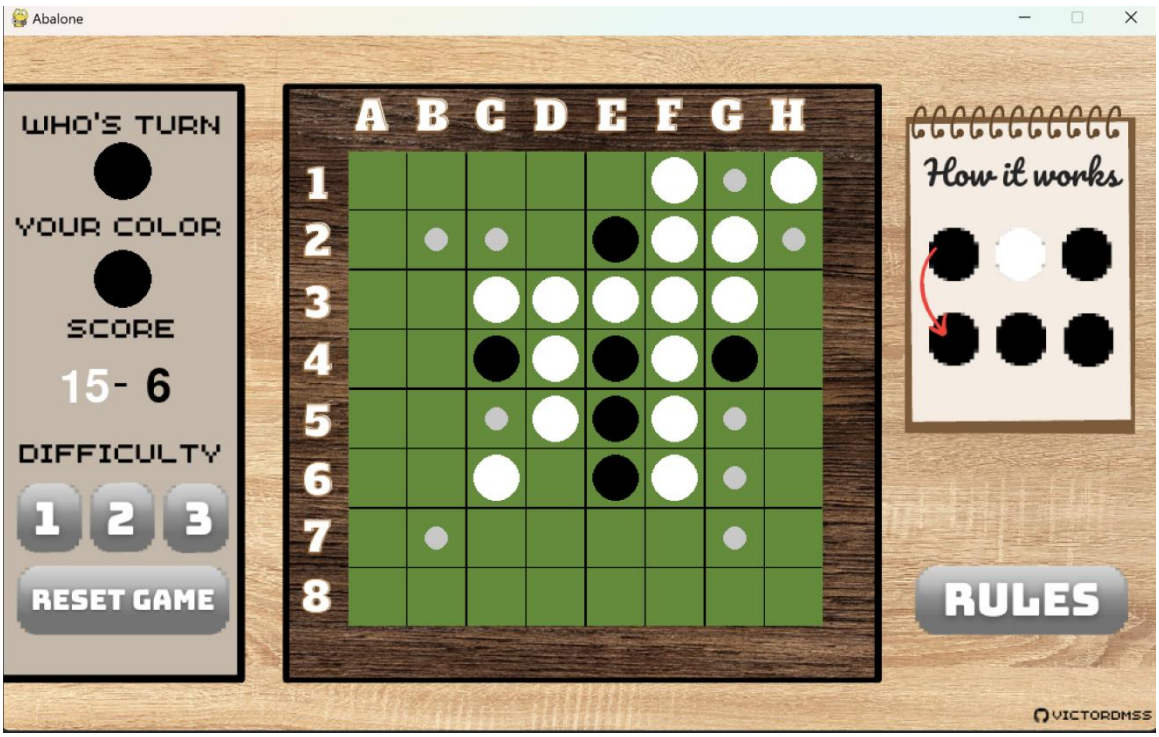
Figure 6: These 2 discs flipped. Discs 1 and 2 are not flipped (even though they appear to be outflanked).

6. All discs outflanked in any one move must be flipped, even if it is to the player's advantage not to flip them at all.

7. Once a disc is placed on a square, it can never be moved to another square later in the game.

8. When the game is over. Discs are counted and the player with the majority of their color showing is the winner.

If the user wants to check the rules during the game, he can directly go on the official website just by clicking on the RULES button (right bottom part of the screen)



LAUNCHING:

The user can choose his color for the whole game. The black color always starts. The other color will be played by an implemented AI (easy level by default).

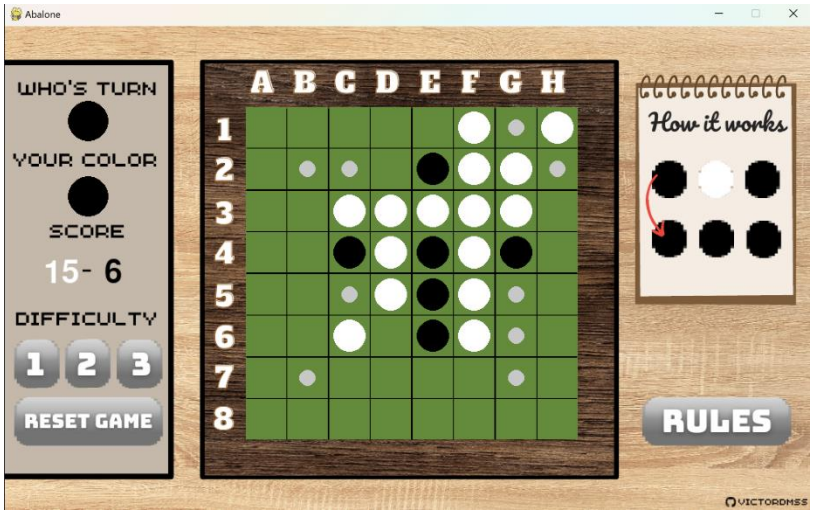
To choose a color, the user just needs to click on one of the two little circles that are on the launch screen.

PLAYING:

The user can see all his available moves with the small grey circles. He just has to click on a valid position on the board to make a move. After that, the AI will play with a little delay and it will be his turn again.

→ *On the left part of the screen, there is some information like the color of the user, the score or the possibility to change the difficulty and reset the game.*

Regarding the difficulty, it can be changed at any time during the game. The modification will be made directly, and the AI will act according to this choice.



ENDING:

When the game ends, the screen changes to show who won the game. However, it remains possible for the user to review the game to understand or to check something. After that, the reset button allows you to play a new game.