AI Report (Mancala using Minimax)

- Team Members and Contribution:

Name	Code	Contribution
اسماعيل طارق اسماعيل رياض	1600271	- **Next_state() Function** - **All code Integration and testing**
عمرو مصطفي عبدالحليم جاب الله	1600943	- **Minimax Algorithm**
مروان محمد محمد زيدان	1601345	- **Game Class** - **S_print()**
محمد شريف عوض محمود محمد المراكبي	1601197	- **Utility_func() function** - **AI class integration**
عبد رب النبي محمد عبدالنبي مصطفي	1600816	- **State class** - **Game_off() function** - **get_actions() function**

- Description (How it works):

The goal of this project is to implement a Mancala game with an AI player using the Minimax algorithm with alpha-beta pruning.

- First, we run the **main** which simply make instance of the **game class** then call the function **set_up()** which sets the initial setup and gets the required information from the user then call the function **game_on()** which runs the game till one players wins

```
from Game import Game

game = Game()  # Create game object
game.set_up()  # Set up game
game.game_on()  # Play game
```

- When **game_on()** is called we go into a **while loop** iterates for whatever number of loops till the **game_off()** function returns **True** which means some player has lost
- When we get into the **game_off()** while loop, we keep iterating between player and AI turns

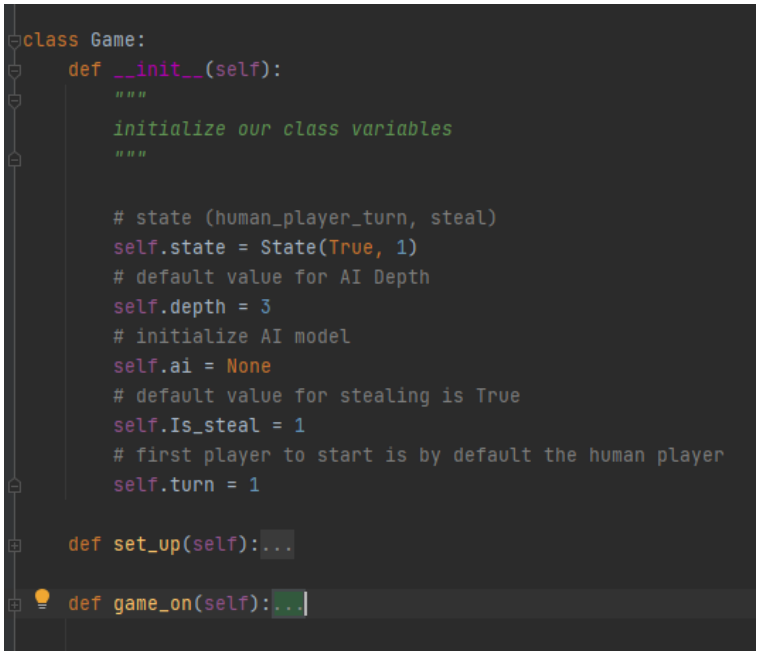
- At each iteration we get action from the **user** and then call **next_state()** function (from **util_file**) to get the next state after executing this action by the player
- Then we do the same for the **AI player** but here comes the **AI class** which is mainly responsible for determination of the Action of the AI player
- We first make an instance of the **AI class** then call the function. **Minimax_alpha_beta_pruning()** which **recursively** builds the tree and using the **utility function ()** returns the best action that could be done by the AI player.
- And keep **alternating** between the two players till one loses.

- Code:

The Code consists of 3 main parts:

A. Game class:

This **class** along with **next_state()** function in **util_file** handle all the **game rules** and it has 3 functions.



```
class Game:
    def __init__(self):
        """
        initialize our class variables
        """

        # state (human_player_turn, steal)
        self.state = State(True, 1)
        # default value for AI Depth
        self.depth = 3
        # initialize AI model
        self.ai = None
        # default value for stealing is True
        self.Is_steal = 1
        # first player to start is by default the human player
        self.turn = 1

    def set_up(self):...

    def game_on(self):...
```

1 – **init() function** to initialize the class attributes we are going to use (**state, depth, ai , is steal, turn**)

2 – **set_up()** function: initialize the setup for the game (get the **difficulty level** and some other information from the user)

```
def set_up(self):
    """
     perform inital setup for the Game
    """
    print(" ==========================    Welcome to Mancala       ==========================")
    self.Is_steal = int(input(" ====== Enter '1' for stealing mode and '0' otherwise =======  : "))
    turn = int(input(" ====== Enter '1' to start first, Enter'0' otherwise ======  : "))
    self.turn = True if turn == 1 else False
    self.state = State(self.turn, self.Is_steal)
    print(" ====== Enter Difficulty Level ====== : ")
    self.depth = 2 + int(input(" (Easy -> 0, Medium -> 1, Hard -> 2, Very hard -> 3) :  )) * 2
    self.ai = AI(self.depth)
    print("Setting up game...")
```

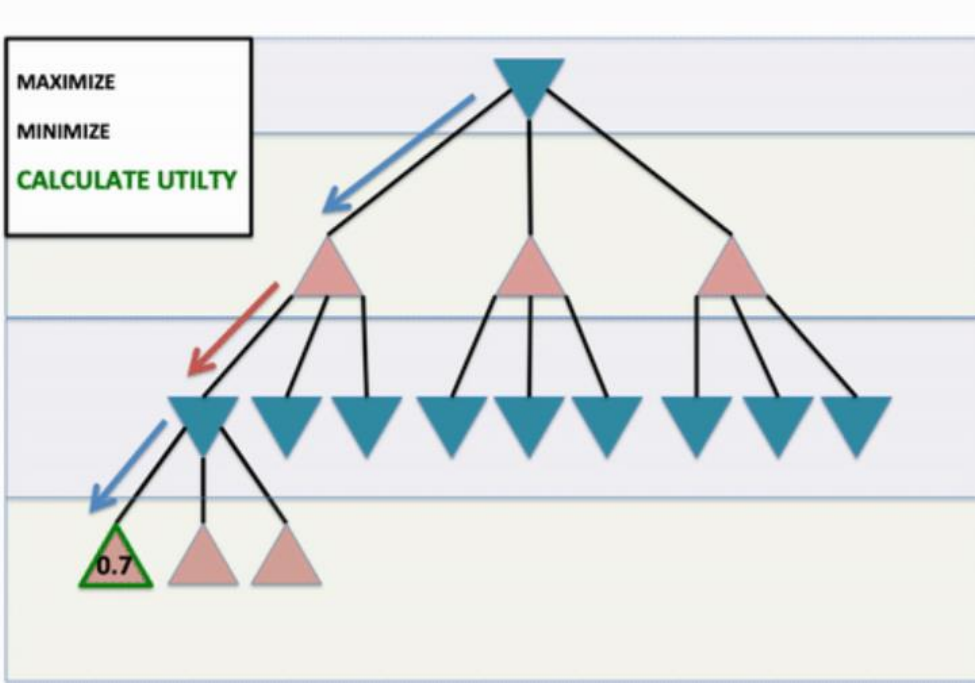
3 – **game_on()** function: starts the game and run till the game is over (any of the players is out of stones)

```
def game_on(self):
    """
    the main function controlling our code, loop until the game is finished and one of the players wins:
     - take the slot number of the player and perform this action
     - perform the AI action returned by our AI algorithm

    :return: new state after each move of a player
    """
    # get initial state of the player
    Util_file.s_print(self.state)
    # loop until the game is finished and one of the players wins
    while not Util_file.game_off(self.state):
        # human_player_turn
        if self.state.human_player_turn:
            slot_number = int(input("Please enter your action : "))
            # get next state after perform this action by calling our util function (next_state)
            self.state = Util_file.next_state(self.state, slot_number)
        # AI Turn
        else:
            # alpha_beta
            # get next state after perform this action by calling our util function
            # (next_state) and the slot number is determined by our mini_max_alpha_beta algorithm
            self.state = Util_file.next_state(self.state, self.ai.minimax_alpha_beta_pruning(self.state))
        # print state after this action
        Util_file.s_print(self.state)
```

B. AI class:

The class responsible for the Minimax algorithm and how to decide the next action for the AI player.



It has 4 functions in addition to is init function

```
class AI:
    """
    a class contain minimax algorithm, and minimax with alpha-beta pruning
    """

    def __init__(self, depth=5):...

    def utility_func(self, state: State):...

    def minimax_alpha_beta_pruning(self, state: State):...

    def max_val(self, state: State, alpha, beta, depth=3):...

    def min_val(self, state: State, alpha, beta, depth=3):...
```

1- The utility function (**utility_func()**) : Our utility function is very simple which is the difference between the AI Player Score and the other player score

```python
def utility_func(self, state: State):
    """
    function to calculate utility for our AI
    Utility is the difference bet. AI player score and Human player Score
    :param state: current state
    :param depth:
    :return: utility value
    """
    utility = state.game_state[13] - state.game_state[6]
    return utility
```

2- **MiniMax_apha_beta_pruning()** function: the main function for AI It generates all possible states in a **recursive** way using the 2 **helper functions (min_val(), max_val())** to consider it as a tree contain all possible states and then when the depth limit is reached the best action is decided based on the utility value returned be utility function

```python
def minimax_alpha_beta_pruning(self, state: State):
    """
    implement minimax algorithm with alpha_beta pruning for our game
    :param state: current state of the game
    :return: the best action to be taken by the AI player given its current state
    """

    utility = []
    alpha = - math.inf
    beta = math.inf
    # loop over each available action and recursively consider it as root to the game tree
    # recursively build the tree of actions to the given depth
    for action in Util_file.get_actions(state):
        # get next state given the current state
        new_state = Util_file.next_state(state, action)
        if new_state.human_player_turn:
            utility.append(
                (self.min_val(new_state, alpha, beta, self.init_depth), action))
        else:
            utility.append((self.max_val(new_state, alpha, beta, self.init_depth), action))

    if state.human_player_turn:
        final_action = min(utility, key=lambda t: t[0])[1]
    else:
        final_action = max(utility, key=lambda t: t[0])[1]

    print("Action chosen by AI: ", final_action)
    return final_action
```

3,4- **min_val(), max_val():** 2 helper function for minimax_alpha_beta algorithm

C. Util_file helper functions: this file contains 4 helper functions to be used in our 2 other classes

1- **Get_actions()** : return the valid actions for the current player, given the current state.

```
def get_actions(state: State):
    """
    get allowed actions for a the player,
    given the current state of the game
    :param state: current state of the game
    :return: allowed actions
    """

    if state.human_player_turn:
        return [i for i, v in enumerate(state.game_state[0:6]) if v > 0]
    else:
        return [i + 7 for i, v in enumerate(state.game_state[7:13]) if v > 0]
```

2- **Game_off()** : decide whether or not the game is over based on the fact that if any of the players is out of stones, the game is over.

```
def game_off(state: State):
    """
    Check if the game is finished or not, calculate the final score

    :param state: the current state of the game
    :return: true if finished, false if not
    """
    # if the human player slots is empty, he has lost
    if sum(state.game_state[0:6]) == 0:
        # calculate the final score (accumulate the remaining stones in the AI player slots to its score)
        state.game_state[13] += sum(state.game_state[7:13])

        return True

    # if the AI player slots is empty, the human player has won
    elif sum(state.game_state[7:13]) == 0:
        # calculate the final score (accumulate the remaining stones in the Human player slots to its score)
        state.game_state[6] += sum(state.game_state[0:6])

        return True

    else:
        return False
```

3- **S_print()**: prints the state of the board every time player makes an action (used in both AI and Game Classes)

```python
def s_print(state: State):
    """
    print the state of the game at a given state
    :param state: state of the game
    """
    # check if the game is finished
    if not game_off(state):
        print("-----------------------------------------------")
        if state.human_player_turn:
            print("    Your turn    ")
        else:
            print("    AI turn    ")
        print("======================")
```

4- **Next_state()**: given a state and an action returns the next state (used in both AI and Game Classes)

- Utility Function:

The basic idea behind the evaluation/ utility function is to give a high value for a board if **maximizer**'s turn or a low value for the board if **minimizer**'s turn.

Our utility function is very simple which is the difference between the AI Player Score and the other player score.

```python
def utility_func(self, state: State):
    """
    function to calculate utility for our AI
    Utility is the difference bet. AI player score and Human player Score
    :param state: current state
    :param depth:
    :return: utility value
    """
    utility = state.game_state[13] - state.game_state[6]
    return utility
```

- Bonus Features:

Support various difficulty levels corresponding to different game tree depths.
We have 4 difficulty levels (Easy, Medium, Hard, very hard)
The formula is depth = 2+2* difficulty level

- User Guide: follow the next steps:

1- To run the code: run Main.py. or run Main.exe.

2- The following will be printed:

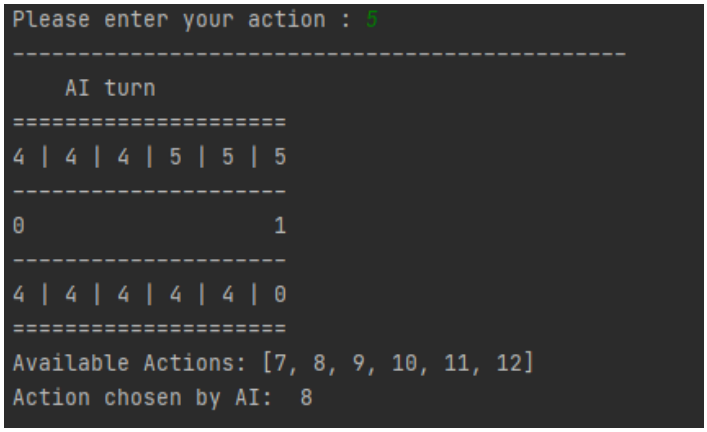
```
 ============================    Welcome to Mancala       ============================
 ====== Enter '1' for stealing mode and '0' otherwise =======  : 1
 ====== Enter '1' to start first, Enter'0' otherwise ======  : 1
 ====== Enter Difficulty Level ====== :
 (Easy -> 0, Medium -> 1, Hard -> 2, Very hard -> 3) :  0
Setting up game...
-----------------------------------------------
```

3- Choose whether to use stealing or not, then whether to play first or second, finally you need to choose the Difficulty level bet (Easy, Medium, Hard, very hard)

4- Then you need to input the action you want to execute given the current state of the game board

```
    Your turn
 ====================
 4 | 4 | 4 | 4 | 4 | 4
 --------------------
 0                  0
 --------------------
 4 | 4 | 4 | 4 | 4 | 4
 ====================
 Available Actions: [0, 1, 2, 3, 4, 5]
 Please enter your action :
```

5- Then comes the AI turn.



6- And so on till the game is over (one of the players is out of stones in all his/her slots), the scoreboard will be printed.