Group members:

Akshat Dubey (234101059)

Vinayak Ichake (234101066)

Implementation of Tic-tac-toe using adversarial search (minimax algorithm)

Problem definition :

Tic-tac-toe is a simple pen and pencil game played between two players. Each of the two players puts noughts ('O') and crosses ('X') in a 3 by 3 grid. The player who first gets 3 consecutive Os or Xs in a row, column or diagonally wins the game and the other player loses. However, if no one is able to achieve this, then it is a draw.

We want to implement a tic-tac-toe game which will be played either between a user and a computer or between two computers. In either case, we want the computer to take the best possible move which will always result in its victory. This optimal move is achieved by using the minimax algorithm.

Input/Data Description :

First, the user is asked a choice whether to play the game between the user and computer or between two computers. If the user selects option 1, then at every time it is user's turn, the user is asked about the location in the grid i.e. x and y coordinate of cell in the grid where he/she wishes to put noughts ('O')/crosses('X'). Here, we have assumed that the first player (computer/user) puts crosses('X') and the second player puts noughts('O').

Solution description / Implementation details :

1. Initial state : Initially, we have assumed that the grid is empty. The grid is stored in the *grid* variable of size 3*3.
2. Finding the score : This is implemented by *utility()* function. It determines whether any player has won or not by searching for consecutive Xs or Os in each row, column and diagonally. If the maximizing player has won, then the score is +1, else if the minimizing player is the winning player, then the score is -1. Otherwise, the score is 0.

3. Selecting the best move / action : For selecting the best move, we are using the minimax algorithm where the maximizing player (first player) puts crosses (‘X’) and the minimizing player (second player) puts noughts (‘O’). First, we are finding all empty locations in the grid using the *get_empty_locations()* function . Then, to find the best move for the current player, we have implemented the *get_best_move*() function. In this, we are traversing all possible game paths through adversarial search i.e. assuming that one player will try to maximize a score while the other players will try to minimize that same score. If it is a maximizing player, the move whichever maximizes the score, will be taken. On the contrary, for a minimizing player, whichever move gives the minimum score, that will be taken.

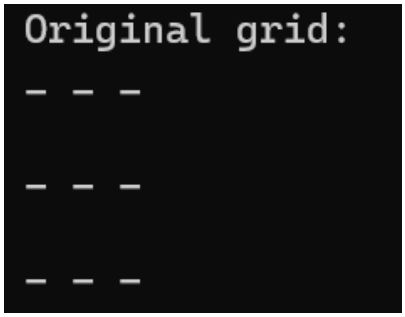
   In the code, we have implemented this adversarial search in *minimax_algorithm()* function.

4. Making the move : It is implemented in the *make_move()* function which just puts ‘X’/’O’ in the location found through the *get_best_move()* function.
5. Printing the grid : After every player’s move, we are printing the updated grid using the *print_grid()* function.
6. These steps are repeated until all empty locations are filled or either of the player wins.

Output/Results :

In the output, we are displaying the updated grid after every player makes a move.

After all empty cells are filled, the status about which player has won or if it is a draw is printed.



Some of the final output for two different iterations of game are given below:

(a) For 1st choice

```
Your turn ('X') :
Enter row number (0/1/2) : 0
Enter column number (0/1/2) : 1
Updated grid :
X X _

X O _

O _ _

Computer's turn ('O'):
Updated grid :
X X O

X O _

O _ _

You have lost! Better luck next time!
```

(b) For 2nd choice

```
Computer 2's turn ('O'):
Updated grid :
X X O

O O X

X O _

Computer 1's turn ('X'):
Updated grid :
X X O

O O X

X O X

It's a draw!
```