Develop a simple AI-Game

(Tic Tac Toe Simple AI Game)

Section : K18AW (Group-2)
Team Members:

1) **Reg No : 11803318 (Ashwani), Roll No : 48**
2) **Reg No : 11802099 (Aman), Roll No : 47**
3) **Reg No : 11802056 (Manas) , Roll No : 46**

Final Report

GitHub : https://github.com/ashwani65/AI-Project

ALGORITHM:

Algo Used : Min-Max Algorithm in Game Theory

Explanation :

play_game() is the main function, which performs following tasks :

- Calls create_board() to create a 9×9 board and initializes with 0.
- For each player (1 or 2), calls the random_place() function to randomly choose a location on board and mark that location with the player number, alternatively.
- Print the board after each move.
- Evaluate the board after each move to check whether a row or column or a diagonal has the same player number. If so, displays the winner name. If after 9 moves, there are no winner then displays “tie”.

Finding the Best Move :

We shall be introducing a new function called **findBestMove()**. This function evaluates all the available moves using **minimax()** and then returns the best move the maximizer can make. The pseudocode is as follows :

```
function findBestMove(board):
    bestMove = NULL
    for each move in board :
        if current move is better than bestMove
            bestMove = current move
    return bestMove
```

Minimax :

To check whether or not the current move is better than the best move we take the help of **minimax()** function which will consider all the possible ways the game can go and returns the best value for that move, assuming the opponent also plays optimally

The code for the maximizer and minimizer in the **minimax()** function is similar to **findBestMove()** , the only difference is, instead of returning a move, it will return a value. Here is the pseudocode :

```
function minimax(board, depth, isMaximizingPlayer):


    if current board state is a terminal state :
        return value of the board


    if isMaximizingPlayer :
        bestVal = -INFINITY
        for each move in board :
            value = minimax(board, depth+1, false)
            bestVal = max( bestVal, value)
        return bestVal
```

```
else :
    bestVal = +INFINITY
    for each move in board :
        value = minimax(board, depth+1, true)
        bestVal = min( bestVal, value)
    return bestVal
```

Checking for GameOver state :

To check whether the game is over and to make sure there are no moves left we use **isMovesLeft()** function. It is a simple straightforward function which checks whether a move is available or not and returns true or false respectively. Pseudocode is as follows :

```
function isMovesLeft(board):
    for each cell in board:
        if current cell is empty:
            return true
    return false
```

Making our AI smarter :

One final step is to make our AI a little bit smarter. Even though the following AI plays perfectly, it might choose to make a move which will result in a slower victory or a faster loss. Lets take an example and explain it.

Assume that there are 2 possible ways for X to win the game from a give board state.

- Move **A** : X can win in 2 move
- Move **B** : X can win in 4 moves

Our evaluation function will return a value of +10 for both moves **A** and **B**. Even though the move **A** is better because it ensures a faster victory, our AI may choose **B** sometimes. To overcome this problem we subtract the depth value from the evaluated score. This means that in case of a victory it will choose a the victory which takes least number of moves and in case of a loss it will try to prolong the game and play as many moves as possible. So the new evaluated value will be

- Move **A** will have a value of +10 – 2 = 8
- Move **B** will have a value of +10 – 4 = 6

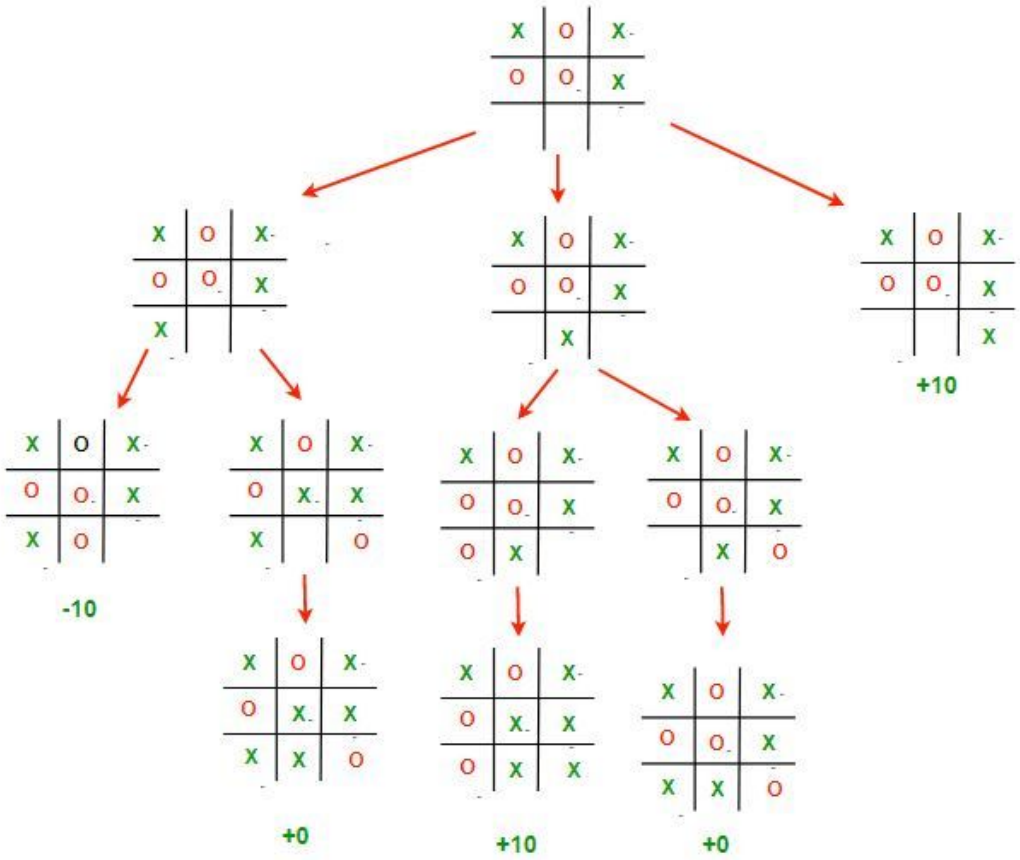
Now since move **A** has a higher score compared to move **B** our AI will choose move **A** over move **B**. The same thing must be applied to the minimizer. Instead of subtracting the depth we add the depth value as the minimizer always tries to get, as negative a value as possible. We can subtract the depth either inside the evaluation function or outside it. Anywhere is fine. I have chosen to do it outside the function. Pseudocode implementation is as follows.

if maximizer has won:

    return WIN_SCORE – depth

else if minimizer has won:

    return LOOSE_SCORE + depth



Output For some Test Cases

Test Case : 1

```
-------------
| 0 | 0 | 0 |
-------------
| 0 | 0 | 0 |
-------------
| 0 | 0 | 0 |
-------------
Input your position:a
-------------
| 0 | 0 | 0 |
-------------
| * | 0 | 0 |
-------------
| x | 0 | 0 |
-------------
Input your position:q
-------------
| * | 0 | 0 |
-------------
| * | 0 | 0 |
-------------
| x | 0 | x |
-------------
Input your position:x
-------------
| * | 0 | 0 |
-------------
| * | 0 | x |
-------------
| x | * | x |
-------------
Input your position:e
-------------
| * | x | * |
-------------
| * | 0 | x |
-------------
| x | * | x |
-------------
Input your position:s
It is a tie!
-------------
| 0 | 0 | 0 |
-------------
| 0 | 0 | 0 |
```

Test Case : 2

```
It is a tie!
-------------
| 0 | 0 | 0 |
-------------
| 0 | 0 | 0 |
-------------
| 0 | 0 | 0 |
-------------
Input your position:q
-------------
| * | 0 | 0 |
-------------
| 0 | x | 0 |
-------------
| 0 | 0 | 0 |
-------------
Input your position:w
-------------
| * | * | x |
-------------
| 0 | x | 0 |
-------------
| 0 | 0 | 0 |
-------------
Input your position:z
-------------
| * | * | x |
-------------
| x | x | 0 |
-------------
| * | 0 | 0 |
-------------
Input your position:d
-------------
| * | * | x |
-------------
| x | x | * |
-------------
| * | x | 0 |
-------------
Input your position:c
It is a tie!
-------------
| 0 | 0 | 0 |
-------------
| 0 | 0 | 0 |
```