NAME –
UID –
BRANCH – B.TECH CSE
SEM – 4TH
SEC- 615 "B"
SUBJECT – AI

Q - SHOW THE WORKING OF MINIMAX ALGORITHM USING TIC TAC TOE GAME..

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
```

CODE IN COMPILER

```
player, opponent = 'x', 'o'


def isMovesLeft(board) :

    for i in range(3) :
        for j in range(3) :
            if (board[i][j] == '_') :
                return True
    return False


def evaluate(b) :

    # Checking for Rows for X or O victory.
    for row in range(3) :
        if (b[row][0] == b[row][1] and b[row][1] == b[row][2]) :
            if (b[row][0] == player) :
                return 10
            else if (b[row][0] == opponent) :
                return -10

    # Checking for Columns for X or O victory.
    for col in range(3) :

        if (b[0][col] == b[1][col] and b[1][col] == b[2][col]) :

            if (b[0][col] == player) :
                return 10
```

```
            else if (b[0][col] == opponent) :
                return -10

    # Checking for Diagonals for X or O victory.
    if (b[0][0] == b[1][1] and b[1][1] == b[2][2]) :

        if (b[0][0] == player) :
            return 10
        else if (b[0][0] == opponent) :
            return -10

    if (b[0][2] == b[1][1] and b[1][1] == b[2][0]) :

        if (b[0][2] == player) :
            return 10
        else if (b[0][2] == opponent) :
            return -10

    # Else if none of them have won then return 0
    return 0


def minimax(board, depth, isMax) :
    score = evaluate(board)


    if (score == 10) :
        return score


    if (score == -10) :
        return score

    # If there are no more moves and no winner then
    # it is a tie
    if (isMovesLeft(board) == False) :
        return 0

    # If this maximizer's move
    if (isMax) :
        best = -1000
```

```
        # Traverse all cells
        for i in range(3) :
            for j in range(3) :

                # Check if cell is empty
                if (board[i][j]=='_') :

                    # Make the move
                    board[i][j] = player

                    # Call minimax recursively and choose
                    # the maximum value
                    best = max( best, minimax(board,
                                              depth + 1,
                                              not isMax) )

                    # Undo the move
                    board[i][j] = '_'
        return best

    # If this minimizer's move
    else :
        best = 1000

        # Traverse all cells
        for i in range(3) :
            for j in range(3) :

                # Check if cell is empty
                if (board[i][j] == '_') :

                    # Make the move
                    board[i][j] = opponent

                    # Call minimax recursively and choose
                    # the minimum value
                    best = min(best, minimax(board, depth + 1, not isMax))

                    # Undo the move
                    board[i][j] = '_'
        return best
```

```
# This will return the best possible move for the player
def findBestMove(board) :
    bestVal = -1000
    bestMove = (-1, -1)

    # Traverse all cells, evaluate minimax function for
    # all empty cells. And return the cell with optimal
    # value.
    for i in range(3) :
        for j in range(3) :

            # Check if cell is empty
            if (board[i][j] == '_') :

                # Make the move
                board[i][j] = player

                # compute evaluation function for this
                # move.
                moveVal = minimax(board, 0, False)

                # Undo the move
                board[i][j] = '_'


                if (moveVal > bestVal) :
                    bestMove = (i, j)
                    bestVal = moveVal

    print("The value of the best Move is :", bestVal)
    print()
    return bestMove
# Driver code
board = [
    [ 'x', 'o', 'x' ],
    [ 'o', 'o', 'x' ],
    [ '_', '_', '_' ]
]

bestMove = findBestMove(board)

print("The Optimal Move is :")
```

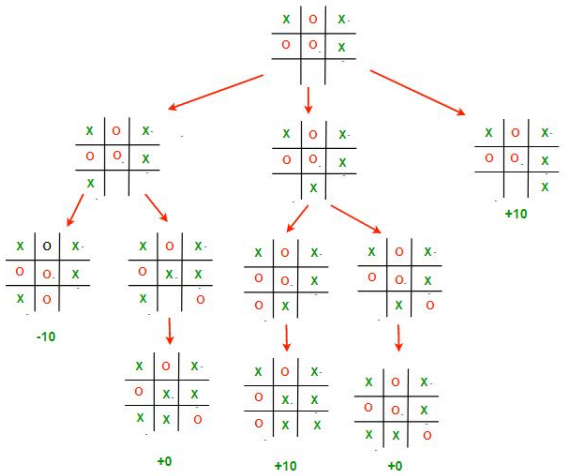
```
print("ROW:", bestMove[0], " COL:", bestMove[1])
```

OUTPUT –

```
rajdeepjaiswal@Rajdeeps-Air py practice % The value of the best Move is : 10

The Optimal Move is :
ROW: 2 COL: 2
```

EXPLAINATION



This image depicts all the possible paths that the game can take from the root board state. It is often called the **Game Tree**.
The 3 possible scenarios in the above example are :

- **Left Move** : If X plays [2,0]. Then O will play [2,1] and win the game. The value of this move is -10
- **Middle Move** : If X plays [2,1]. Then O will play [2,2] which draws the game. The value of this move is 0
- **Right Move** : If X plays [2,2]. Then he will win the game. The value of this move is +10;

Remember, even though X has a possibility of winning if he plays the middle move, O will never let that happen and will choose to draw instead.
Therefore the best choice for X, is to play [2,2], which will guarantee a victory for him.
We do encourage our readers to try giving various inputs and understanding why the AI choose to play that move. Minimax may confuse programmers as it it thinks several moves in advance and is very hard to debug at times. Remember this implementation of minimax algorithm can be applied any 2 player board game with

some minor changes to the board structure and how we iterate through the moves. Also sometimes it is impossible for minimax to compute every possible game state for complex games like Chess. Hence we only compute upto a certain depth and use the evaluation function to calculate the value of the board.
Stay tuned for next weeks article where we shall be discussing about **Alpha-Beta pruning**that can drastically improve the time taken by minimax to traverse a game tree.