Aman Kumar Pandey
RA1911003010685
Artificial Intelligence Lab
Lab - 6

Aim: Implementation of minimax algorithm for an application

Problem Formulation

Consider a board having nine elements vector where each element will contain '-' for blank, x for indicating the move of player 1 and o for player 2's move.

Initial State

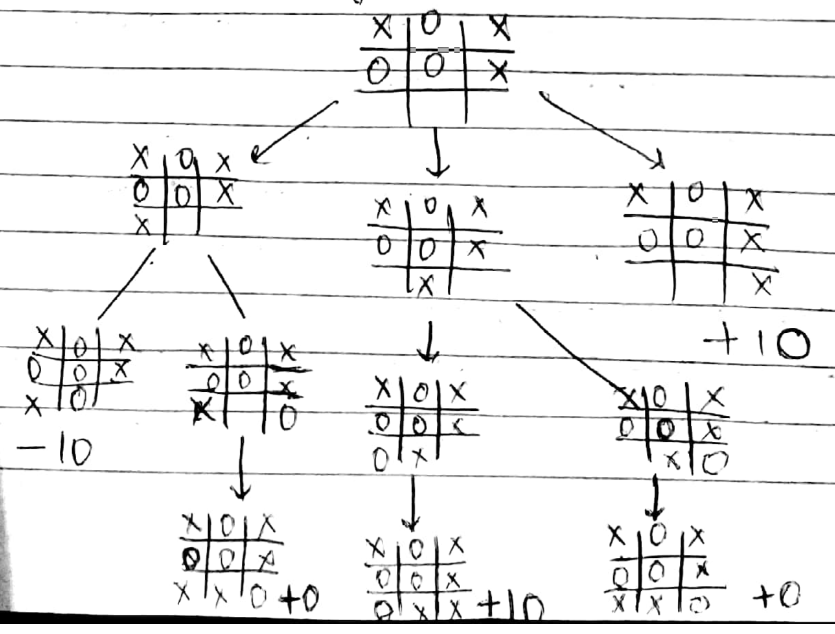
X	O	X
O	O	X
-	-	-

Final State

X	O	X
O	O	X
O	X	X

+10

Problem Solving



if player 1 play [2, 2] then he will win the game.
The value of this move is +10

AMAN KUMAR PANDEY

RA1911003010685

ARTIFICIAL INTELLIGENCE LAB

EXPERIMENT NO: 6

IMPLEMENTATION OF MINIMAX ALGORITHM FOR AN APPLICATION

Algorithm:

Step-1: Start

Step-2: Construct the complete game tree

Step-3: Evaluate scores for leaves using the evaluation function

Step-4: Back-up scores from leaves to root, considering the player type:

- For max player, select the child with the maximum score
- For min player, select the child with the minimum score

Step-5: At the root node, choose the node with max value and perform the corresponding move

Step-6: Stop

Source code:

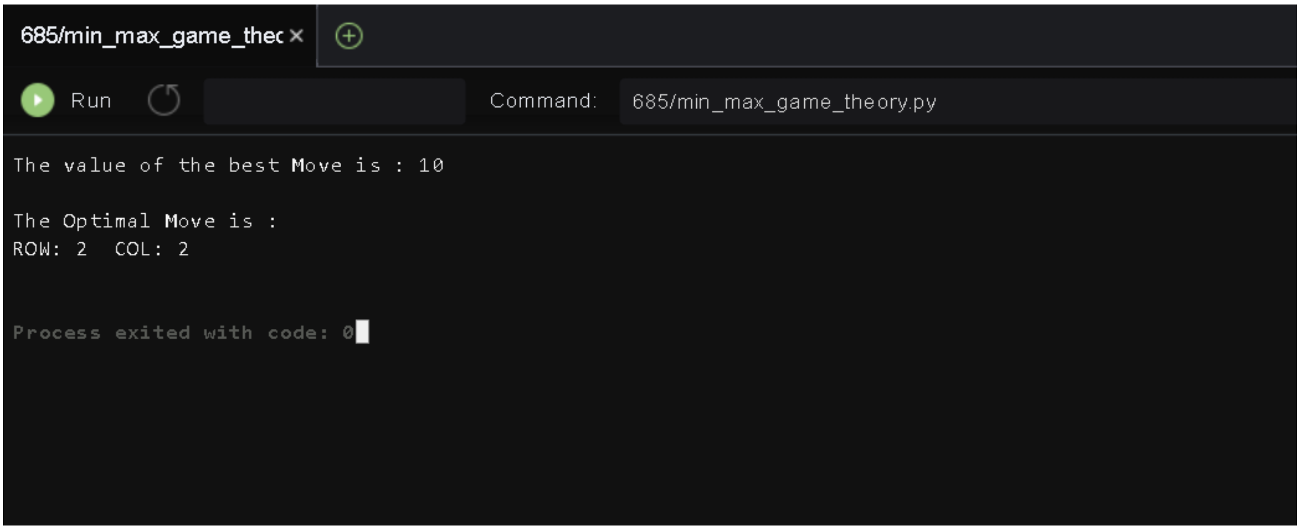
```
 # Python3 program to find the next optimal move for a player
player, opponent = 'x', 'o'

# This function returns true if there are moves
# remaining on the board. It returns false if
# there are no moves left to play.
def isMovesLeft(board) :
    for i in range(3) :
        for j in range(3) :
            if (board[i][j] == '_'):
                return True
    return False
# This is the evaluation function as discussed
# in the previous article ( http://goo.gl/sJgv68 )
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
            elif (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
            elif (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
        elif (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
        elif (b[0][2] == opponent) :
            return -10
    # Else if none of them have won then return 0
    return 0
# This is the minimax function. It considers all
# the possible ways the game can go and returns
# the value of the board
def minimax(board, depth, isMax) :
    score = evaluate(board)
    # If Maximizer has won the game return his/her
    # evaluated score
    if (score == 10) :
        return score
    # If Minimizer has won the game return his/her
    # evaluated 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 the value of the current move is
                # more than the best value, then update
                # best/
                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



RESULT

Hence, the Implementation of the minimax algorithm for TIC-TAC-TOE is done successfully.