Department of Computer Science and Engineering (Data Science)

Subject: Artificial Intelligence (DJS22DSC502)

AY: 2024-25

Experiment 7

DHRUV SHAH 60009220132 D1-1

(Adversarial Search)

Aim: Implement Tic-Tac-Toe game using Minimax algorithm.

Theory:

- Mini-max algorithm is a recursive or backtracking algorithm which is used in decision-making and game theory. It provides an optimal move for the player assuming that opponent is also playing optimally.
- Mini-Max algorithm uses recursion to search through the game-tree.
- Min-Max algorithm is mostly used for game playing in AI. Such as Chess, Checkers, tic-tac-toe, go, and various tow-players game. This Algorithm computes the minimax decision for the current state.
- In this algorithm two players play the game, one is called MAX and other is called MIN.
- Both the players fight it as the opponent player gets the minimum benefit while they get the maximum benefit.
- Both Players of the game are opponent of each other, where MAX will select the maximized value and MIN will select the minimized value.
- The minimax algorithm performs a depth-first search algorithm for the exploration of the complete game tree.
- The minimax algorithm proceeds all the way down to the terminal node of the tree, then backtrack the tree as the recursion.

Psudo Code

```
1. function minimax(node, depth, maximizingPlayer) is
2. if depth ==0 or node is a terminal node then
3. return static evaluation of node
4. if MaximizingPlayer then      // for Maximizer Player
5. maxEva= -infinity
6.  for each child of node do
7.  eva= minimax(child, depth-1, false)
8. maxEva= max(maxEva,eva)       //gives Maximum of the values
9. return maxEva
10. else                         // for Minimizer player
11.  minEva= +infinity
12.  for each child of node do
13.  eva= minimax(child, depth-1, true)
14.  minEva= min(minEva, eva)       //gives minimum of the values
15.  return minEva
```

Initial call:

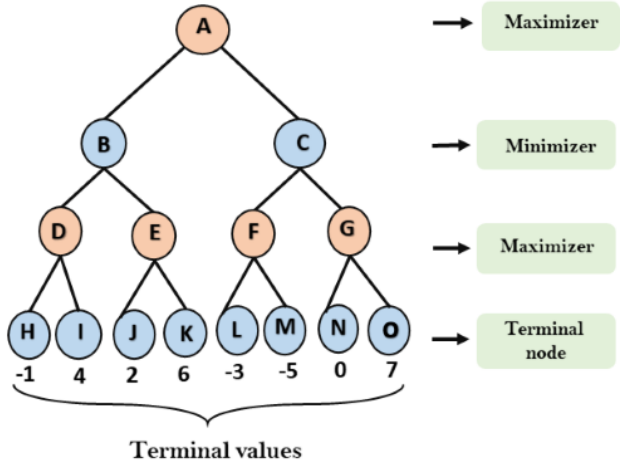
Minimax(node, 3, true)

Working of Min-Max Algorithm:

- The working of the minimax algorithm can be easily described using an example. Below we have taken an example of game-tree which is representing the two-player game.
- In this example, there are two players one is called Maximizer and other is called Minimizer.
- Maximizer will try to get the Maximum possible score, and Minimizer will try to get the minimum possible score.
- This algorithm applies DFS, so in this game-tree, we have to go all the way through the leaves to reach the terminal nodes.

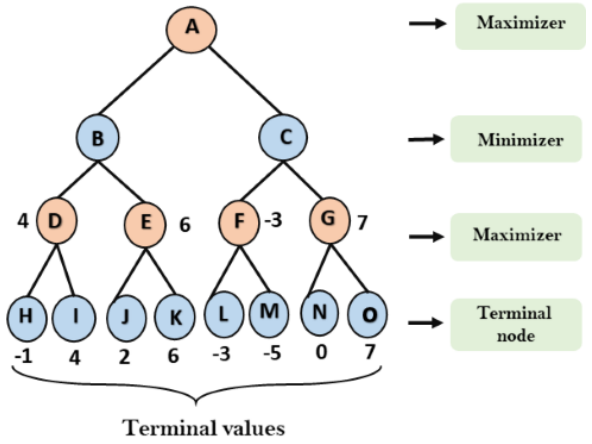
- At the terminal node, the terminal values are given so we will compare those value and backtrack the tree until the initial state occurs. Following are the main steps involved in solving the two-player game tree:

Step-1: In the first step, the algorithm generates the entire game-tree and apply the utility function to get the utility values for the terminal states. In the below tree diagram, let's take A is the initial state of the tree. Suppose maximizer takes first turn which has worst-case initial value =- infinity, and minimizer will take next turn which has worst-case initial value = +infinity.



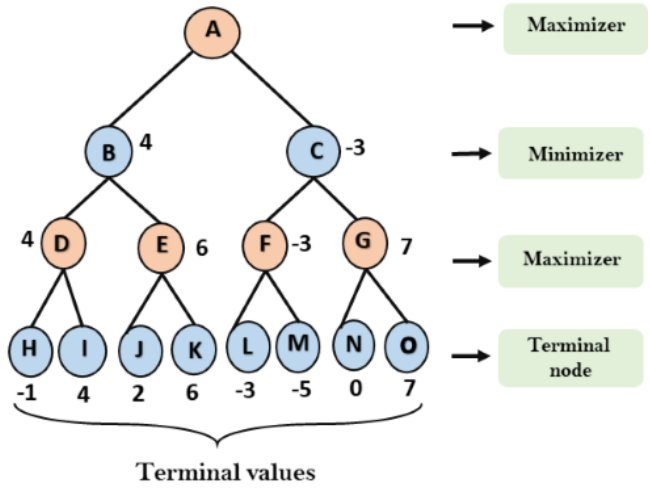
Step 2: Now, first we find the utilities value for the Maximizer, its initial value is -∞, so we will compare each value in terminal state with initial value of Maximizer and determines the higher nodes values. It will find the maximum among the all.

- For node D max(-1,- -∞) => max(-1,4)= 4
- For Node E max(2, -∞) => max(2, 6)= 6
- For Node F max(-3, -∞) => max(-3,-5) = -3
- For node G max(0, -∞) = max(0, 7) = 7



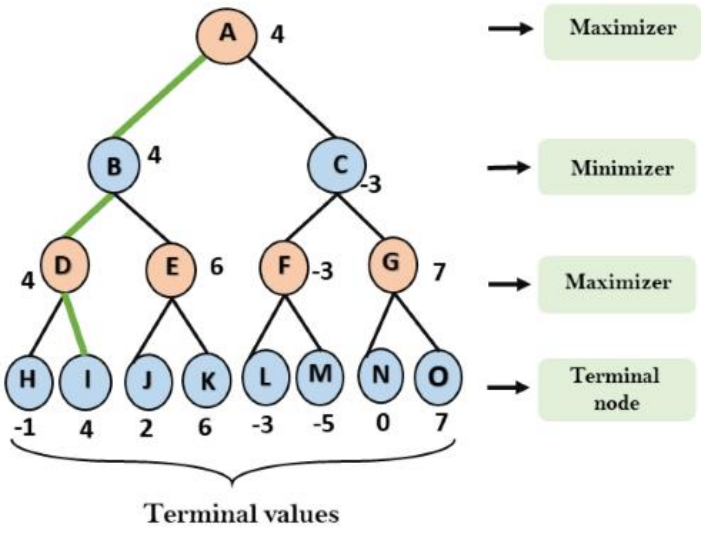
Step 3: In the next step, it's a turn for minimizer, so it will compare all nodes value with +∞, and will find the 3rd layer node values.

- For node B= min(4,6) = 4
- For node C= min (-3, 7) = -3



Step 4: Now it's a turn for Maximizer, and it will again choose the maximum of all nodes value and find the maximum value for the root node. In this game tree, there are only 4 layers, hence we reach immediately to the root node, but in real games, there will be more than 4 layers.

- For node A max(4, -3)= 4



Lab Assignment to do:

Implement a two player Tic-Tac-Toe game using Minimax algorithm for Game Playing. One player will be computer itself.

DHRUV SHAH

```
import random, numpy as np, pandas as pd


Comp = 'X'
You = 'O'
empty = ' '


def output(board):
    for row in board:
        print(row)
    print()


def check_winner(board):
    for i in range(3):
        if board[i][0] == board[i][1] == board[i][2] != empty:
            return board[i][0]
        if board[0][i] == board[1][i] == board[2][i] != empty:
            return board[0][i]

    if board[0][0] == board[1][1] == board[2][2] != empty:
        return board[0][0]
    if board[0][2] == board[1][1] == board[2][0] != empty:
        return board[0][2]

    return None


def is_full(board):
    for row in board:
        for cell in row:
            if cell == empty:
                return False
    return True


def score_evaluation(board):
    winner = check_winner(board)
    if winner == 'X':
        return -10
    elif winner == 'O':
        return 10
    else:
        return 0

board = [['X', 'O', ''], ['O', 'X', 'O'], ['X', '', 'X']]
score_evaluation(board)
```

```
-10
```

```
def minimax(board, depth, maximizingPlayer):
    winner = check_winner(board)

    if winner == Comp:
        return -10 + depth
    if winner == You:
        return 10 - depth
    if is_full(board):
        return 0

    if maximizingPlayer:
        maxEva = -np.inf
        for i in range(3):
            for j in range(3):
                if board[i][j] == empty:
                    board[i][j] = You
                    eva = minimax(board, depth + 1, False)
                    board[i][j] = empty
                    maxEva = max(maxEva, eva)
        return maxEva
    else:
        minEva = np.inf
        for i in range(3):
            for j in range(3):
                if board[i][j] == empty:
                    board[i][j] = Comp
```

```
                    eva = minimax(board, depth + 1, True)
                    board[i][j] = empty
                    minEva = min(minEva, eva)
        return minEva


def comp_move(board):
    best_value = -np.inf
    best_move = (-1, -1)

    for i in range(3):
        for j in range(3):
            if board[i][j] == empty:
                board[i][j] = You
                move_value = minimax(board, 0, False)
                board[i][j] = empty

                if move_value > best_value:
                    best_move = (i, j)
                    best_value = move_value

    return best_move


board = np.array([[empty, empty, empty],
                  [empty, empty, empty],
                  [empty, empty, empty]])
board[0][0] = Comp
board[0][1] = You
board[1][0] = Comp
board[1][1] = You

print("Current Board:")
output(board)

winner = check_winner(board)
print("Winner:", winner)

full = is_full(board)
print("Is the board full?", full)

score = score_evaluation(board)
print("Score Evaluation:", score)

best_move = comp_move(board)
print("Best move for the computer:", best_move)
```

```
Current Board:
['X' 'O' ' ']
['X' 'O' ' ']
[' ' ' ' ' ']

Winner: None
Is the board full? False
Score Evaluation: 0
Best move for the computer: (2, 1)
```

```
def main():
    board = [[' ', ' ', ' '], [' ', ' ', ' '], [' ', ' ', ' ']]
    current_player = You
    minimax_final_score = None

    while True:
        output(board)

        if current_player == You:
            while True:
                user_input = input("Enter your move (row and column): ")
                try:
                    row, col = map(int, user_input.split())
                    if board[row][col] == empty:
                        board[row][col] = You
                        break
                    else:
                        print("Cell already occupied! Try again.")
                except (ValueError, IndexError):
                    print("Invalid input! Enter row and column as two numbers between 0 and 2.")
        else:
            print("Computer's turn:")
            row, col = comp_move(board)
            board[row][col] = Comp

        winner = check_winner(board)
```

```
            if winner:
                output(board)
                print(f"The winner is: {winner}")
                minimax_final_score = minimax(board, 0, winner == You)
                print("Minimax score at the end:", minimax_final_score)
                break
            elif is_full(board):
                output(board)
                print("The game is a draw!")
                minimax_final_score = 0
                print("Minimax score at the end: 0")
                break

            current_player = Comp if current_player == You else You


    if __name__ == "__main__":
        main()
```

```
[' ', ' ', ' ']
[' ', ' ', ' ']
[' ', ' ', ' ']

Enter your move (row and column): 2 0
[' ', ' ', ' ']
[' ', ' ', ' ']
['O', ' ', ' ']

Computer's turn:
['X', ' ', ' ']
[' ', ' ', ' ']
['O', ' ', ' ']

Enter your move (row and column): 1 1
['X', ' ', ' ']
[' ', 'O', ' ']
['O', ' ', ' ']

Computer's turn:
['X', ' ', 'X']
[' ', 'O', ' ']
['O', ' ', ' ']

Enter your move (row and column): 2 2
['X', ' ', 'X']
[' ', 'O', ' ']
['O', ' ', 'O']

Computer's turn:
['X', ' ', 'X']
[' ', 'O', ' ']
['O', 'X', 'O']

Enter your move (row and column): 1 0
['X', ' ', 'X']
['O', 'O', ' ']
['O', 'X', 'O']

Computer's turn:
['X', ' ', 'X']
['O', 'O', 'X']
['O', 'X', 'O']

Enter your move (row and column): 0 1
['X', 'O', 'X']
['O', 'O', 'X']
['O', 'X', 'O']

The game is a draw!
Minimax score at the end: 0
```

Start coding or generate with AI.