Shri Vile Parle Kelavani Mandal's
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC Accredited with "A" Grade (CGPA : 3.18)

Department of Computer Science and Engineering (Data Science)

**Subject: Artificial Intelligence (DJ19DSC502)**

**AY: 2022 - 23**

**Experiment 7**

**(Adversarial**

**Search)**

**Name: Bhuvi Ghosh** **SAPID: 60009210191** **Batch:** D22

**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:

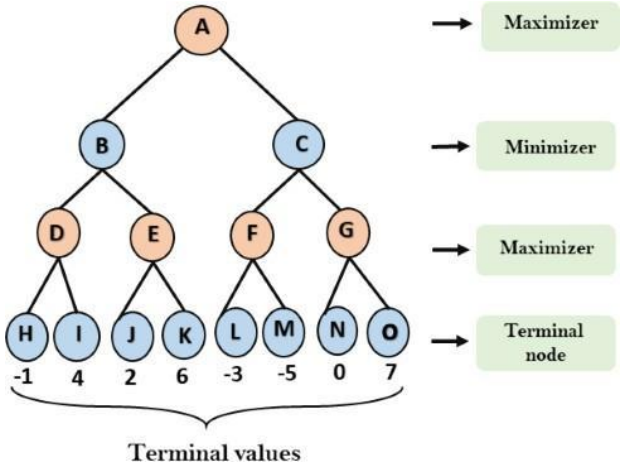
Minimax(node, 3, true)

**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.

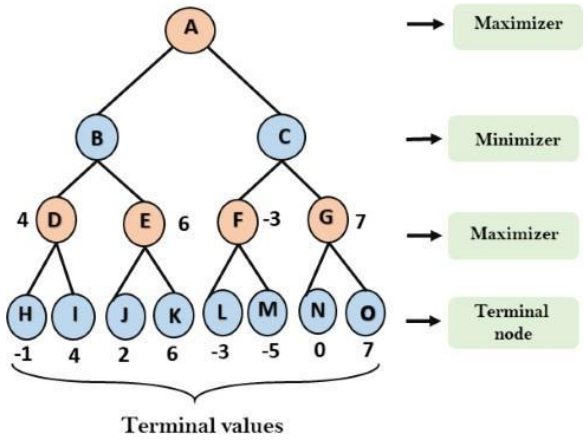
- At the terminal node, the terminal values are given so we will compare those value and backtrack the tree until the initial state occurs. Following are the main steps involved in solving the two-player game tree:

**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.



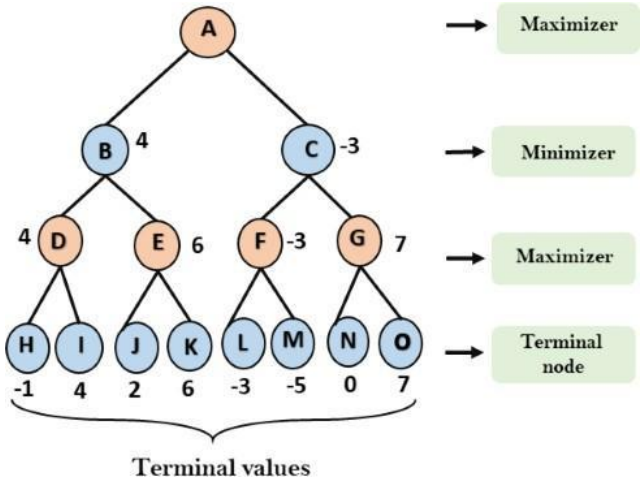

**Step 2:** Now, first we find the utilities value for the Maximizer, its initial value is -∞, so we will compare each value in terminal state with initial value of Maximizer and determines the higher nodes values. It will find the maximum among the all.

- 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



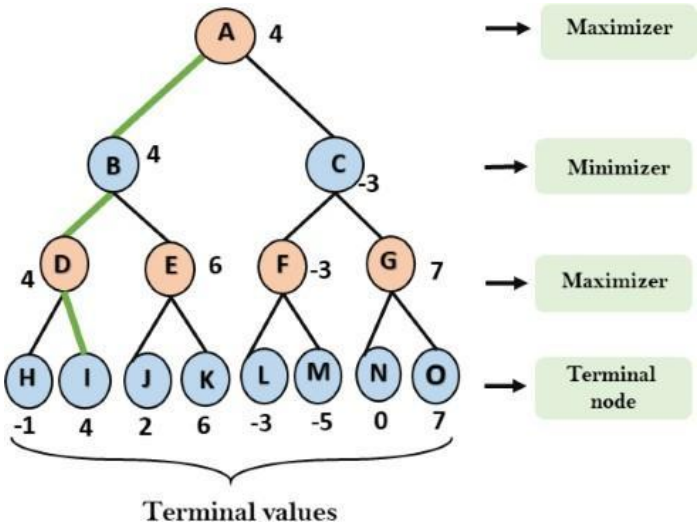

**Step 3:** In the next step, it's a turn for minimizer, so it will compare all nodes value with +∞, and will find the 3rd layer node values.

- 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.

Code:

```
import math

class TicTacToe:
    def __init__(self):
        self.board = [' ' for _ in range(9)]
        self.current_winner = None

    def print_board(self):
        for row in [self.board[i*3:(i+1)*3] for i in range(3)]:
            print('| ' + ' | '.join(row) + ' |')

    @staticmethod
    def print_board_nums():
        number_board = [[str(i) for i in range(j * 3, (j + 1) * 3)] for j in range(3)]
        for row in number_board:
            print('| ' + ' | '.join(row) + ' |')

    def available_moves(self):
        return [i for i, spot in enumerate(self.board) if spot == ' ']

    def empty_squares(self):
        return ' ' in self.board

    def num_empty_squares(self):
        return self.board.count(' ')
```

```
    def make_move(self, square, letter):
        if self.board[square] == ' ':
            self.board[square] = letter
            if self.winner(square, letter):
                self.current_winner = letter
            return True
        return False

    def winner(self, square, letter):

        row_ind = square // 3
        row = self.board[row_ind*3:(row_ind+1)*3]
        if all([spot == letter for spot in row]):
            return True

        col_ind = square % 3
        column = [self.board[col_ind+i*3] for i in range(3)]
        if all([spot == letter for spot in column]):
            return True

        if square % 2 == 0:
            diagonal1 = [self.board[i] for i in [0, 4, 8]]
            if all([spot == letter for spot in diagonal1]):
                return True
            diagonal2 = [self.board[i] for i in [2, 4, 6]]
            if all([spot == letter for spot in diagonal2]):
                return True
```

```
def play(game, x_player, o_player, print_game=True):
    if print_game:
        game.print_board_nums()

    letter = 'X'
    while game.empty_squares():
        if letter == 'O':
            square = o_player.get_move(game)
        else:
            square = x_player.get_move(game)

        if game.make_move(square, letter):
            if print_game:
                print(f'{letter} makes a move to square {square}')
                game.print_board()
                print('')

            if game.current_winner:
                if print_game:
                    print(f'{letter} wins!')
                return letter
            letter = 'O' if letter == 'X' else 'X'

        if print_game:
            print('It\'s a tie!')
```

```
class HumanPlayer:
    def __init__(self, letter):
        self.letter = letter

    def get_move(self, game):
        valid_square = False
        val = None
        while not valid_square:
            square = input(f'{self.letter}\'s turn. Choose a square (0-8): ')
            try:
                val = int(square)
                if val not in game.available_moves():
                    raise ValueError
                valid_square = True
            except ValueError:
                print('Invalid square. Try again.')
        return val
```

```
class AIPlayer:
    def __init__(self, letter):
        self.letter = letter

    def get_move(self, game):
        if len(game.available_moves()) == 9:
            square = 4
        else:
            square = self.minimax(game, self.letter)['position']
        return square

    def minimax(self, state, player):
        max_player = self.letter
        other_player = 'O' if player == 'X' else 'X'

        if state.current_winner == other_player:
            return {'position': None, 'score': 1 * (state.num_empty_squares() + 1) if other_player == max_player else -1 * (state.num_empty_squares() + 1)}
        elif not state.empty_squares():
            return {'position': None, 'score': 0}

        if player == max_player:
            best = {'position': None, 'score': -math.inf}
        else:
            best = {'position': None, 'score': math.inf}
```

```
        for possible_move in state.available_moves():
            state.make_move(possible_move, player)
            sim_score = self.minimax(state, other_player)

            state.board[possible_move] = ' '
            state.current_winner = None
            sim_score['position'] = possible_move

            if player == max_player:
                if sim_score['score'] > best['score']:
                    best = sim_score
            else:
                if sim_score['score'] < best['score']:
                    best = sim_score

        return best

if __name__ == '__main__':
    x_player = HumanPlayer('X')
    o_player = AIPlayer('O')
    t = TicTacToe()
    play(t, x_player, o_player, print_game=True)
```

Output:

```
| 0 | 1 | 2 |
| 3 | 4 | 5 |
| 6 | 7 | 8 |
X's turn. Choose a square (0-8): 0
X makes a move to square 0
| X |   |   |
|   |   |   |
|   |   |   |

It's a tie!
O makes a move to square 4
| X |   |   |
|   | O |   |
|   |   |   |

It's a tie!
X's turn. Choose a square (0-8): 8
X makes a move to square 8
| X |   |   |
|   | O |   |
|   |   | X |

It's a tie!
O makes a move to square 1
| X | O |   |
|   | O |   |
|   |   | X |

It's a tie!
X's turn. Choose a square (0-8): 7
X makes a move to square 7
| X | O |   |
|   | O |   |
|   | X | X |
```

```
It's a tie!
O makes a move to square 6
| X | O |   |
|   | O |   |
| O | X | X |

It's a tie!
X's turn. Choose a square (0-8): 2
X makes a move to square 2
| X | O | X |
|   | O |   |
| O | X | X |

It's a tie!
O makes a move to square 5
| X | O | X |
|   | O | O |
| O | X | X |

It's a tie!
X's turn. Choose a square (0-8): 3
X makes a move to square 3
| X | O | X |
| X | O | O |
| O | X | X |

It's a tie!
```