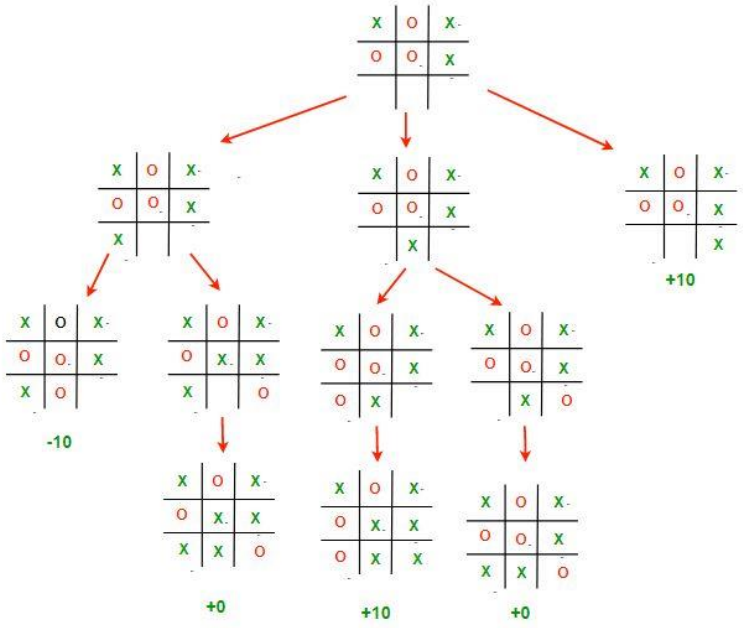
Name	**Hatim Yusuf Sawai**
UID no.	**2021300108**
Experiment No.	**4**

AIM:	**Informed Searching using Min-Max Algorithm**
PROBLEM DEFINITION:	**Implement a Tic Tac Toe problem using the Informed searching technique min-max algorithm . Analyze the algorithm with respect to Completeness, Optimality, time and space Complexity**
THEORY:	**MinMax Searching:** The Minimax algorithm is a decision-making algorithm commonly used in two-player games, such as chess, checkers, and tic-tac-toe, to determine the best move for a player while considering the opponent's moves. It's a fundamental technique in game theory and artificial intelligence for adversarial games. The goal of the Minimax algorithm is to find the best strategy that minimizes the maximum possible loss (hence "minimax") or maximizes the minimum possible gain, assuming that both players play optimally. **Properties of Min-Max Search:** **Complete?** – Yes (if tree is finite) **Optimal?** – Yes (against an optimal opponent) – No (does not exploit opponent weakness against suboptimal opponent) **Time complexity?** – $O(b^m)$ **Space complexity?** – O(bm) (depth-first exploration)

Solving Tic-tac-toe using MinMax Search:

Using the Minimax algorithm to solve Tic-Tac-Toe involves constructing a game tree that explores every possible sequence of moves and opponent responses. At each node of the tree, it selects the move that either maximizes or minimizes the outcome based on the current player's perspective. Terminal nodes are assigned values representing wins, losses, or draws. These values are propagated back up the tree following the Minimax principle. Ultimately, the algorithm identifies the best move for the current player, ensuring optimal decisions in the game. It's a systematic way to play Tic-Tac-Toe, aiming for victory or a draw, considering all potential moves and counter moves by the opponent.



CODE:

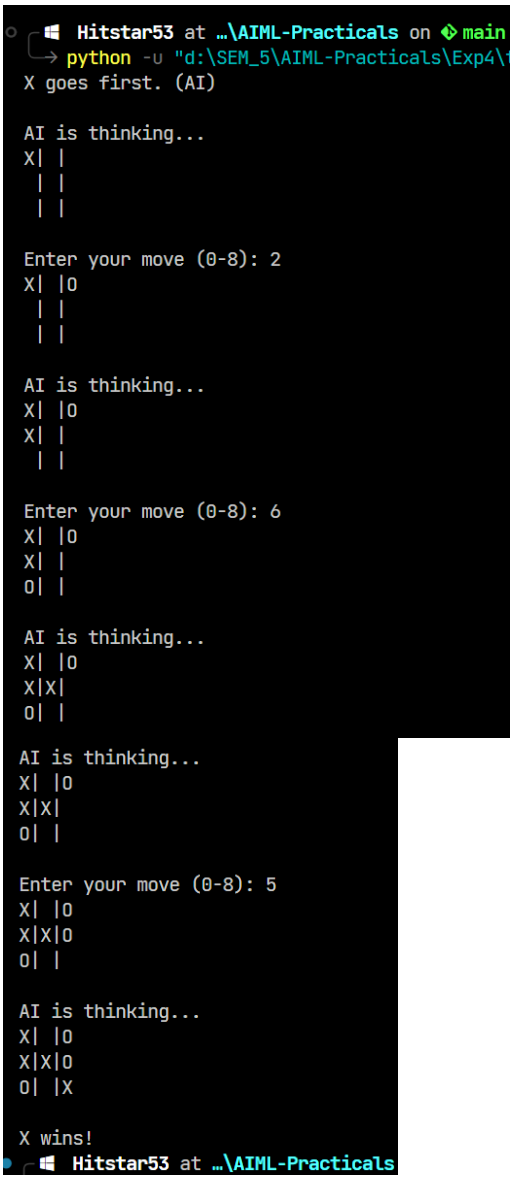
```
import random

class TicTacToe:
    def __init__(self):
        self.board = [' ' for _ in range(9)]
        # pick x/o using random
        if random.randint(0, 1) == 0:
            self.current_player = 'X'
            print("X goes first. (AI)\n")
        else:
            self.current_player = 'O'
            print("O goes first. (You)\n")

    def print_board(self):
        for i in range(0, 9, 3):
            print('|'.join(self.board[i:i+3]))

    def is_full(self):
        return ' ' not in self.board

    def is_winner(self, player):
        winning_combinations = [(0, 1, 2), (3, 4, 5), (6, 7, 8),
                                (0, 3, 6), (1, 4, 7), (2, 5, 8),
                                (0, 4, 8), (2, 4, 6)]
        for a, b, c in winning_combinations:
            if self.board[a] == self.board[b] == self.board[c] == player:
                return True
        return False

    def is_terminal(self):
        return self.is_winner('X') or self.is_winner('O') or self.is_full()

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

    def max_value(self, alpha, beta):
```

```
        if self.is_terminal():
            if self.is_winner('X'):
                return 1
            elif self.is_winner('O'):
                return -1
            else:
                return 0

        max_eval = float('-inf')
        for cell in self.get_empty_cells():
            self.board[cell] = 'X'
            eval = self.min_value(alpha, beta)
            self.board[cell] = ' '
            max_eval = max(max_eval, eval)
            alpha = max(alpha, eval)
            if beta <= alpha:
                break
        return max_eval

    def min_value(self, alpha, beta):
        if self.is_terminal():
            if self.is_winner('X'):
                return 1
            elif self.is_winner('O'):
                return -1
            else:
                return 0

        min_eval = float('inf')
        for cell in self.get_empty_cells():
            self.board[cell] = 'O'
            eval = self.max_value(alpha, beta)
            self.board[cell] = ' '
            min_eval = min(min_eval, eval)
            beta = min(beta, eval)
            if beta <= alpha:
```

```
            break
        return min_eval

    def best_move(self):
        best_move = None
        best_eval = float('-inf')
        alpha = float('-inf')
        beta = float('inf')
        for cell in self.get_empty_cells():
            self.board[cell] = 'X'
            eval = self.min_value(alpha, beta)
            self.board[cell] = ' '
            if eval > best_eval:
                best_eval = eval
                best_move = cell
        return best_move

    def play(self):
        while not self.is_terminal():
            if self.current_player == 'X':
                print("AI is thinking...")
                move = self.best_move()
                self.board[move] = 'X'
                self.current_player = 'O'
            else:
                move = int(input("Enter your move (0-8): "))
                if self.board[move] == ' ':
                    self.board[move] = 'O'
                    self.current_player = 'X'
                else:
                    print("Invalid move. Try again.")
            self.print_board()
            print()

        if self.is_winner('X'):
            print("X wins!")
```

```
        elif self.is_winner('O'):
            print("O wins!")
        else:
            print("It's a draw!")

if __name__ == "__main__":
    game = TicTacToe()
    game.play()
```

OUTPUT:



CONCLUSION:	In this experiment, we implemented Tic-Tac-Toe using the Minimax algorithm and conducted a comprehensive analysis. The Minimax algorithm exhibited completeness by exhaustively searching for solutions and demonstrated its optimality in strategic decision-making. We also considered the algorithm's time and space complexity, highlighting its efficiency in solving adversarial games.