Introduction to AI

MSE -1

Tic Tac Toe Solver

Name – Sneh singh

Branch – CSE AI/ML

Section – C

Semester – 2

INTRODUCTION

This project is a **Tic-Tac-Toe Solver** that uses the **Minimax Algorithm** to make optimal moves for the AI player. Tic-Tac-Toe is a simple two-player game where players take turns placing their mark (either 'X' or 'O') on a 3x3 grid. The objective is to get three of their marks in a row, either horizontally, vertically, or diagonally.

While Tic-Tac-Toe is a game that can be played manually with little effort, the purpose of this program is to demonstrate how an AI can play the game optimally by considering all possible moves and evaluating the best option at each turn. The **Minimax algorithm** is a decision-making method often used in two-player games, where the AI simulates all possible moves and selects the one that maximizes its chances of winning while minimizing the opponent's chances.

METHODOLOGY

1. Representation of the Tic-Tac-Toe Board

The Tic-Tac-Toe board is represented as a 3x3 grid, where each cell can either be empty or contain a player's mark ('X' or 'O'). The board is implemented using a list of lists or a 1D list for simplicity. Each cell can hold one of three values:

- 'X' for Player 1 (AI or Human)
- 'O' for Player 2 (opponent)
- ' ' (empty space) for an unfilled position

The board state is continuously updated after each move, and the program checks for game-ending conditions after every turn.

2. Minimax Algorithm

The heart of the AI's decision-making process is the **Minimax algorithm**. The algorithm follows a recursive approach to explore all possible future moves and assigns a score to each state based on the result:

- **+1** if the AI (Player 1) wins
- **-1** if the opponent (Player 2) wins

- **0** if the game ends in a draw

Minimax Process:

- **Base Case**: When the game reaches a terminal state (win, loss, or draw), the algorithm returns a score.
    - If the AI wins, the score is **+1**.
    - If the opponent wins, the score is **-1**.
    - If the game ends in a draw, the score is **0**.
- **Recursive Case**: The algorithm recursively simulates all possible moves for both players. For each move, it evaluates the game state after that move and calls the minimax function again for the next move.
    - If it's the **AI's turn** (maximizing player), it chooses the move that leads to the highest score.
    - If it's the **opponent's turn** (minimizing player), it chooses the move that leads to the lowest score.
- **Pruning**: The Minimax algorithm works by evaluating all possible moves and recursively simulating the game state until the game reaches a terminal condition. **Alpha-Beta Pruning** can be implemented to optimize the algorithm by eliminating branches of

the game tree that won't influence the final decision, thus reducing the number of calculations required.

3. Evaluating Possible Moves

For each possible move, the algorithm simulates the new board state, evaluates the outcome using the Minimax algorithm, and then selects the move that maximizes (or minimizes) the score based on whether it's the AI's or the opponent's turn. Here's how moves are evaluated:

1. **Generate all valid moves**: Identify all possible empty spaces on the board where a move can be made.
2. **Simulate each move**: For each valid move, place the respective player's mark and call the Minimax function recursively to evaluate the resulting board state.
3. **Score Calculation**: After evaluating all potential moves, the algorithm picks the one with the highest (for maximizing player) or lowest (for minimizing player) score.

4. Game State Evaluation

The program needs to check for **terminal game states** after each move. This includes:

- **Win Condition**: If a player has three marks in a row (horizontally, vertically, or diagonally).
- **Draw Condition**: If the board is full and no player has won.

If any of these conditions are met, the game ends, and the result (win/loss/draw) is returned.

5. User Input and AI Interaction

The program alternates turns between the user and the AI:

- The user provides input for their move, entering the row and column where they wish to place their mark (either 'X' or 'O').
- The AI uses the Minimax algorithm to calculate the optimal move based on the current board state and plays its turn.

6. Game Loop

The game runs in a continuous loop until a winner is determined or the game ends in a draw. After each move, the game checks the board state for a win or draw condition, and if none is met, the game continues with the next player's turn.

7. Termination and Output

The game terminates when:

- The user or AI wins by completing a line of three marks.
- The game ends in a draw if there are no empty spaces and no winner.

The final board state is printed after each move, and the result (win/loss/draw) is displayed at the end of the game.

CODE

```
import math

# Define the board size
BOARD_SIZE = 3

# Player markers
PLAYER_X = 'X'
PLAYER_O = 'O'
EMPTY = ' '

# Function to check if the game is over (win or draw)
def check_winner(board):
    # Check rows, columns, and diagonals for a winner
    for i in range(BOARD_SIZE):
        # Check rows and columns
        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]
```

```
    # Check diagonals
    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]

    # Check if there's a draw (no more empty spaces)
    if all(board[i][j] != EMPTY for i in range(BOARD_SIZE) for j in
range(BOARD_SIZE)):
        return "Draw"

    # The game is not over yet
    return None

# Function to evaluate the board for the minimax algorithm
def evaluate(board):
    winner = check_winner(board)

    # Return a score based on the winner
    if winner == PLAYER_X:
        return 1  # X wins
    elif winner == PLAYER_O:
```

```
        return -1  # O wins
    elif winner == "Draw":
        return 0  # Draw

    return 0  # Game is ongoing or no winner yet

# Minimax algorithm to find the best move for the player
def minimax(board, depth, is_maximizing_player):
    score = evaluate(board)

    # If the current state is a terminal state, return the score
    if score != 0:
        return score

    # Maximizing player (X)
    if is_maximizing_player:
        best = -math.inf

        # Try every possible move for X
        for i in range(BOARD_SIZE):
            for j in range(BOARD_SIZE):
                if board[i][j] == EMPTY:
```

```
                board[i][j] = PLAYER_X  # Make the move
                best = max(best, minimax(board, depth + 1, False))  # Minimize for O
                board[i][j] = EMPTY  # Undo the move

        return best

    # Minimizing player (O)
    else:
        best = math.inf

        # Try every possible move for O
        for i in range(BOARD_SIZE):
            for j in range(BOARD_SIZE):
                if board[i][j] == EMPTY:
                    board[i][j] = PLAYER_O  # Make the move
                    best = min(best, minimax(board, depth + 1, True))  # Maximize for X
                    board[i][j] = EMPTY  # Undo the move

        return best

# Function to find the best move for the current player (X)
```

```
def find_best_move(board):
    best_val = -math.inf
    best_move = (-1, -1)

    # Try every possible move for X
    for i in range(BOARD_SIZE):
        for j in range(BOARD_SIZE):
            if board[i][j] == EMPTY:
                board[i][j] = PLAYER_X  # Make the move
                move_val = minimax(board, 0, False)  # Get the score for the
move
                board[i][j] = EMPTY  # Undo the move

                # If the current move is better than the best move, update it
                if move_val > best_val:
                    best_move = (i, j)
                    best_val = move_val

    return best_move

# Function to print the board (for visualization purposes)
def print_board(board):
```

```
    for row in board:
        print(" | ".join(row))
        print("-" * 5)

# Function to check if the move is valid
def is_valid_move(board, row, col):
    return 0 <= row < BOARD_SIZE and 0 <= col < BOARD_SIZE and board[row][col] == EMPTY

# Function to play the Tic-Tac-Toe game
def play_game():
    # Initial empty board
    board = [
        [EMPTY, EMPTY, EMPTY],
        [EMPTY, EMPTY, EMPTY],
        [EMPTY, EMPTY, EMPTY]
    ]

    print("Welcome to Tic-Tac-Toe!")
    print("You are O and the computer is X.")
    print("Let's start!")
```

```
    # Display the initial empty board
    print_board(board)

    # Main game loop
    while True:
        # Player's move (O)
        while True:
            try:
                row, col = map(int, input("\nEnter your move (row and column from 0 to 2): ").split())
                if is_valid_move(board, row, col):
                    board[row][col] = PLAYER_O
                    break
                else:
                    print("Invalid move. Try again.")
            except ValueError:
                print("Invalid input. Please enter two integers separated by a space.")

        # Display the board after player's move
        print("\nYour move:")
        print_board(board)
```

```
# Check if the game is over
winner = check_winner(board)
if winner:
    if winner == "Draw":
        print("\nThe game is a draw!")
    else:
        print(f"\n{winner} wins!")
    break

# Computer's move (X)
print("\nComputer's move:")
best_move = find_best_move(board)
board[best_move[0]][best_move[1]] = PLAYER_X

# Display the board after computer's move
print_board(board)

# Check if the game is over
winner = check_winner(board)
if winner:
    if winner == "Draw":
        print("\nThe game is a draw!")
```

```
            else:
                print(f"\n{winner} wins!")
            break

# Run the game
if __name__ == "__main__":
    play_game()
```

Output

```
Welcome to Tic-Tac-Toe!
You are O and the computer is X.
Let's start!
  |   |
-----
  |   |
-----
  |   |
-----

Enter your move (row and column from 0 to 2): 1 2

Your move:
  |   |
-----
  |   | O
-----
  |   |
-----
```

```
You are O and the computer is X.
Let's start!
  |   |
-----
  |   |
-----
  |   |
-----

Enter your move (row and column from 0 to 2): 1 2

Your move:
  |   |
-----
  |   | O
-----
  |   |
-----

Computer's move:
X |   |
-----
  |   | O
-----
  |   |
```

```
-----

Enter your move (row and column from 0 to 2): 2 2

Your move:
X |   |
-----
  |   | O
-----
  |   | O
-----

Computer's move:
X |   | X
-----
  |   | O
-----
  |   | O
-----

Enter your move (row and column from 0 to 2): 2 1

Your move:
X |   | X
-----
  |   | O
-----
  | O | O
```

```
Computer's move:
X | X | X
-----
  |   | O
-----
  | O | O
-----

X wins!
```