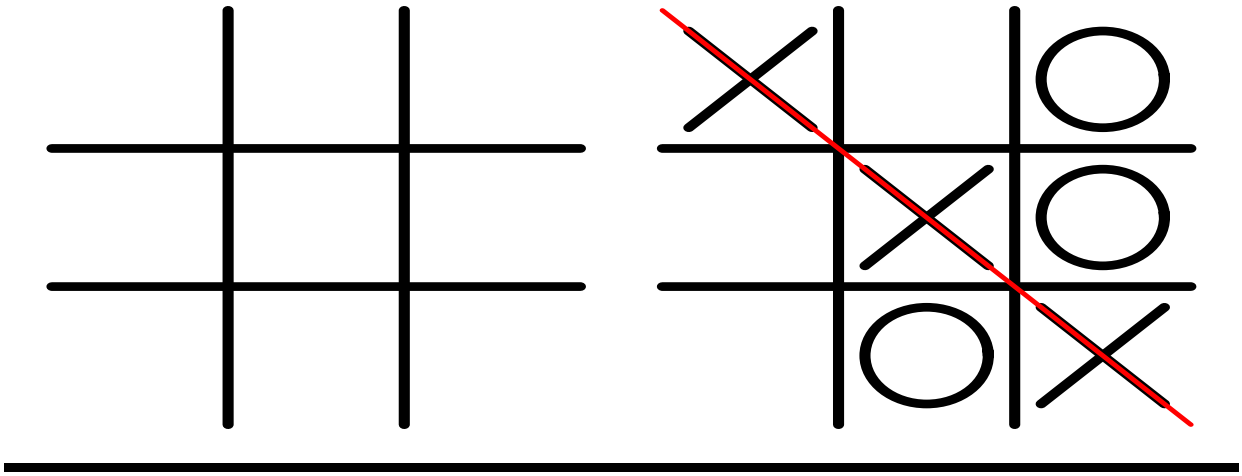
Tic-Tac-Toe Solver



Submitted By -

Mayank Chaudhary

Roll No-5

CSE(AI) - C

Tic-Tac-Toe Solver

INTRODUCTION

- The project, *Tic-Tac-Toe Solver*, is a Python-based game where a human competes against an AI opponent.
- It involves the development of an AI capable of making optimal moves using decision-making algorithms.
- The AI employs strategies like the Minimax algorithm to analyse the game state and ensure challenging gameplay.
- This project highlights concepts of game logic, artificial intelligence, and Python programming.
- It provides an engaging way to explore AI implementation and user interaction in a simple yet strategic game.

METHODOLOGY

- Define Objective: Create a Python game where a human plays against an AI in Tic-Tac-Toe.
- Choose Algorithm: Implement the Minimax algorithm for optimal AI moves.

- Design Game: Develop a 3x3 grid, symbols (X/O), and turn-based gameplay.
- Build AI: Code AI to evaluate moves, handle edge cases, and ensure challenging play.
- Create Interface: Design a simple interface for interaction and results display.
- Test & Debug: Ensure smooth gameplay by identifying and fixing issues.
- Document: Finalize the project and prepare detailed documentation.

Tic-Tac-Toe Solver

CODE

```
#Optimal Decisions in games

import math


# Constants for the players

AI = "X"  # AI is the maximizing player

HUMAN = "O"  # Human is the minimizing player

EMPTY = " "  # Empty cell in the board
```

```
# Function to print the Tic-Tac-Toe board
def print_board(board):
    for row in board:
        print("|".join(row))
    print("\n")

# Check if a player has won
def check_winner(board):
    # All possible winning combinations: rows, columns, diagonals
    win_combinations = [
        [board[0][0], board[0][1], board[0][2]],  # Row 1
        [board[1][0], board[1][1], board[1][2]],  # Row 2
        [board[2][0], board[2][1], board[2][2]],  # Row 3
        [board[0][0], board[1][0], board[2][0]],  # Column 1
        [board[0][1], board[1][1], board[2][1]],  # Column 2
        [board[0][2], board[1][2], board[2][2]],  # Column 3
        [board[0][0], board[1][1], board[2][2]],  # Diagonal 1
        [board[0][2], board[1][1], board[2][0]]   # Diagonal 2
    ]

    # Check for a winner
    if [AI, AI, AI] in win_combinations: return AI
    if [HUMAN, HUMAN, HUMAN] in win_combinations: return HUMAN
    return None  # No winner yet

# Check if the game is a draw
def is_draw(board):
    for row in board:
        if EMPTY in row:  # If there are any empty spots
            return False  # Not a draw yet
    return True  # All spots are filled

# MiniMax Algorithm to find the best move
def minimax(board, is_maximizing):
```

```
    winner = check_winner(board)
    if winner == AI: return 1  # AI wins, return 1
    if winner == HUMAN: return -1  # Human wins, return -1
    if is_draw(board): return 0  # Draw, return 0

    if is_maximizing:  # AI's turn (maximize score)
        best_score = -math.inf  # Start with a very low score
        for i in range(3):
            for j in range(3):
                if board[i][j] == EMPTY:  # If the cell is empty
                    board[i][j] = AI  # Try placing AI's symbol
                    score = minimax(board, False)  # Recursively evaluate the move
                    board[i][j] = EMPTY  # Undo the move
                    best_score = max(best_score, score)  # Maximize score for AI
        return best_score
    else:  # Human's turn (minimize score)
        best_score = math.inf  # Start with a very high score
        for i in range(3):
            for j in range(3):
                if board[i][j] == EMPTY:  # If the cell is empty
                    board[i][j] = HUMAN  # Try placing Human's symbol
                    score = minimax(board, True)  # Recursively evaluate the move
                    board[i][j] = EMPTY  # Undo the move
                    best_score = min(best_score, score)  # Minimize score for Human
        return best_score

# Find the best move for AI
def find_best_move(board):
    best_score = -math.inf
    best_move = None
    for i in range(3):
        for j in range(3):
            if board[i][j] == EMPTY:  # If the cell is empty
                board[i][j] = AI  # Try placing AI's symbol
```

```
            score = minimax(board, False)  # Get the score for the move
            board[i][j] = EMPTY  # Undo the move
            if score > best_score:  # If this move is better than the previous one
                best_score = score
                best_move = (i, j)  # Save the best move
    return best_move

# Function for the user to input their move
def user_move(board):
    while True:
        try:
            move = int(input("Enter your move (1-9): ")) - 1  # User inputs a number from 1-9
            row, col = divmod(move, 3)  # Convert the input to board indices
            if board[row][col] == EMPTY:  # Check if the cell is empty
                board[row][col] = HUMAN  # Place the Human's symbol
                break
            else:
                print("Cell already occupied. Try again.")
        except (ValueError, IndexError):
            print("Invalid move! Please enter a number from 1-9 corresponding to an empty cell.")

# Example Tic-Tac-Toe board (3x3 grid)
board = [
    [" ", " ", " "],
    [" ", " ", " "],
    [" ", " ", " "]
]

# Print the current board
print("Welcome to Tic-Tac-Toe!")
print_board(board)

# Main game loop
while True:
```

```
# Player (Human) move
user_move(board)
print("Your Move:")
print_board(board)

if check_winner(board):
    print("Congratulations, you win!")
    break
if is_draw(board):
    print("It's a draw!")
    break

# AI move (Optimal Move)
print("AI is making a move...")
best_move = find_best_move(board)
if best_move:
    board[best_move[0]][best_move[1]] = AI  # AI makes the move
    print("AI's Move:")
    print_board(board)
else:
    print("Game Over! No moves left.")
    break

if check_winner(board):
    print("AI wins!")
    break
if is_draw(board):
    print("It's a draw!")
    break
```

OUTPUT / RESULT