**Tic-Tac-Toe Solver**

**Introduction**

Tic-Tac-Toe is a classic two-player game where players take turns to mark a 3x3 grid with 'X' and 'O'. The objective is to align three of one's marks in a row, either horizontally, vertically, or diagonally. This report discusses a Tic-Tac-Toe solver using the **Minimax algorithm**, enabling an AI (playing 'O') to play optimally against a human player (playing 'X').

**Objective**

The goal of the project is to create an interactive game where:

* The player plays as 'X' and the AI plays as 'O'.
* The AI uses the **Minimax algorithm** to determine the optimal move, ensuring that it either wins or forces a draw.
* The game ends when either player wins, or the board is full, resulting in a draw.

**Minimax Algorithm**

The **Minimax algorithm** is a decision-making strategy used in turn-based games. It evaluates all possible moves, assuming both players are playing optimally:

* **Maximizing Player (AI)**: The AI tries to maximize its chances of winning.
* **Minimizing Player (Player)**: The player tries to minimize the AI's chances of winning.

The algorithm recursively evaluates game states, assigning scores based on whether the AI or the player wins, and uses **alpha-beta pruning** to optimize performance.

**Code Overview**

The game is implemented with:

* A 3x3 grid to represent the board.
* The **Minimax algorithm** to evaluate all possible moves and select the optimal one.
* Alpha-beta pruning to optimize the decision-making process by eliminating irrelevant moves.

The game alternates between the player and the AI, updating the board after each move and checking for a winner or a draw.

**Gameplay Example**

In the game, the player ('X') can win if they strategically place their marks to form a winning line. The AI ('O') will always play optimally, either winning or forcing a draw. A typical game flow involves the player making a move, followed by the AI calculating its best response.

**Conclusion**

The Tic-Tac-Toe solver using the **Minimax algorithm** ensures that the AI always plays optimally, either winning or drawing. The human player must think strategically to win or avoid a loss. This solution demonstrates how game theory can be applied to create intelligent decision-making in games.

**Future Work**

Future improvements include:

* Adding a graphical user interface (GUI) for better user interaction.
* Introducing different difficulty levels for the AI.
* Expanding the algorithm to more complex games with larger search spaces.

**CODE**

import random

# Initialize the board

def initialize\_board():

    return [[' ' for \_ in range(3)] for \_ in range(3)]

# Display the board

def print\_board(board):

    for row in board:

        print('|'.join(row))

        print('-' \* 5)

# Check if the current board has a winner

def check\_winner(board, player):

    # Check rows, columns, and diagonals for a win

    for i in range(3):

        if all([board[i][j] == player for j in range(3)]):  # Check row

            return True

        if all([board[j][i] == player for j in range(3)]):  # Check column

            return True

    # Check diagonals

    if all([board[i][i] == player for i in range(3)]):  # Left diagonal

        return True

    if all([board[i][2-i] == player for i in range(3)]):  # Right diagonal

        return True

    return False

# Check if the board is full (draw)

def check\_draw(board):

    for row in board:

        for cell in row:

            if cell == ' ':

                return False

    return True

# Get all available empty positions on the board

def get\_empty\_cells(board):

    empty\_cells = []

    for i in range(3):

        for j in range(3):

            if board[i][j] == ' ':

                empty\_cells.append((i, j))

    return empty\_cells

# Minimax algorithm for the best move

def minimax(board, depth, is\_maximizing, alpha, beta):

    # Check if the game has ended

    if check\_winner(board, 'X'):

        return -10 + depth  # X wins

    if check\_winner(board, 'O'):

        return 10 - depth  # O wins

    if check\_draw(board):

        return 0  # Draw

    # Maximize O's score (AI)

    if is\_maximizing:

        max\_eval = float('-inf')

        for row, col in get\_empty\_cells(board):

            board[row][col] = 'O'  # Make the move

            eval = minimax(board, depth + 1, False, alpha, beta)

            board[row][col] = ' '  # Undo the move

            max\_eval = max(max\_eval, eval)

            alpha = max(alpha, eval)

            if beta <= alpha:

                break

        return max\_eval

    else:

        # Minimize X's score (player)

        min\_eval = float('inf')

        for row, col in get\_empty\_cells(board):

            board[row][col] = 'X'  # Make the move

            eval = minimax(board, depth + 1, True, alpha, beta)

            board[row][col] = ' '  # Undo the move

            min\_eval = min(min\_eval, eval)

            beta = min(beta, eval)

            if beta <= alpha:

                break

        return min\_eval

# Find the best move for the AI (O)

def find\_best\_move(board):

    best\_move = None

    best\_value = float('-inf')

    for row, col in get\_empty\_cells(board):

        board[row][col] = 'O'

        move\_value = minimax(board, 0, False, float('-inf'), float('inf'))

        board[row][col] = ' '

        if move\_value > best\_value:

            best\_value = move\_value

            best\_move = (row, col)

    return best\_move

# Play the game

def play\_game():

    board = initialize\_board()

    while True:

        print\_board(board)

        # Player (X) move

        row, col = map(int, input("Enter your move (X) (row col): ").split())

        if board[row][col] != ' ':

            print("Invalid move! Try again.")

            continue

        board[row][col] = 'X'

        if check\_winner(board, 'X'):

            print\_board(board)

            print("Player X wins!")

            break

        if check\_draw(board):

            print\_board(board)

            print("It's a draw!")

            break

        # AI (O) move

        print("AI is making its move...")

        ai\_move = find\_best\_move(board)

        if ai\_move:

            board[ai\_move[0]][ai\_move[1]] = 'O'

            if check\_winner(board, 'O'):

                print\_board(board)

                print("AI (O) wins!")

                break

            if check\_draw(board):

                print\_board(board)

                print("It's a draw!")

                break

# Start the game

play\_game()

**OUTPUT**

|X| | |

-----

| |X| |

-----

| | | |

Enter your move (row col): 1 1

|X| | |

-----

| |X| |

-----

| | | |

AI is making its move...

|X| | |

-----

| |X|O|

-----

| | | |

Enter your move (row col): 2 0

|X| | |

-----

| |X|O|

-----

|X| | |

AI is making its move...

|X| | |

-----

|O|X|O|

-----

|X| | |

Enter your move (row col): 0 2

|X| |X|

-----

|O|X|O|

-----

|X| | |

AI is making its move...

|X| |X|

-----

|O|X|O|

-----

|X|O| |

Enter your move (row col): 2 2

|X| |X|

-----

|O|X|O|

-----

|X|O|X|

Player X wins!