Report on Tic-Tac-Toe Solver in Python

1. Introduction

Tic-Tac-Toe is a classic two-player game often used to teach basic programming concepts. The game involves a 3x3 grid, and each player alternates placing their mark (usually an "X" or "O") on the grid. The objective is to place three of the player's marks in a row, either horizontally, vertically, or diagonally.
A **Tic-Tac-Toe solver** is a program that can determine the optimal move in any given board state or can even predict the winner of the game given a particular configuration. The game can either end in a win for one of the players or a draw if no player wins.
This report explains the approach to creating a Tic-Tac-Toe solver using Python, covering the game mechanics, the algorithm used, and the implementation details.

2. Problem Definition

The task of solving a Tic-Tac-Toe game involves two main objectives:

1. **Determine if a player has won**: This is determined by checking if any row, column, or diagonal contains three identical marks ("X" or "O").
2. **Predict optimal moves**: Using an algorithm like the **Minimax algorithm**, the solver should predict the best possible move for the current player, considering the opponent's optimal moves.

3. Game Mechanics

A Tic-Tac-Toe game typically follows these steps:

1. **Board Setup**: A 3x3 grid where each cell is either empty, contains an "X", or contains an "O".
2. **Turns**: Two players take alternate turns to place their mark ("X" or "O") on an empty cell.
3. **Win Condition**: The game ends when one player places three of their marks consecutively in a row, column, or diagonal, or if all cells are filled with no winner, resulting in a draw.
4. **End Game**: The game ends when there is a winner or all cells are filled (draw).

4. Solving Approach

To solve the Tic-Tac-Toe game programmatically, we need to:

- **Evaluate the board**: The solver should be able to evaluate if a given board is in a winning or draw state.

- **Simulate moves**: The solver must simulate different board configurations resulting from possible moves and predict the winner.

- **Optimal Strategy**: Implement an algorithm like **Minimax** that helps choose the best move by simulating future moves and choosing the one that maximizes the player's chances of winning while minimizing the opponent's chances.
- **Minimax Algorithm**:
- The **Minimax algorithm** is a decision-making algorithm often used in two-player games like Tic-Tac-Toe. The core idea is:

1. **Maximizing Player (X)**: The player who is trying to win will try to maximize their score (taking the best possible move).

2. **Minimizing Player (O)**: The opponent will try to minimize the score of the maximizing player (taking the best move to block or prevent a win).

The algorithm simulates all possible moves and chooses the one that maximizes the chance of winning for the maximizing player while minimizing the opponent's potential.

Steps of Minimax:

- Evaluate the board recursively.
- Assign a score for each possible end state (win, loss, draw).
- Propagate the scores back through the decision tree.
- Select the optimal move based on the maximum score for the current player.

Implementation

The implementation of the Tic-Tac-Toe solver involves creating a function to evaluate the board, a function to implement the Minimax algorithm, and a function to determine the best move based on the current board.

CODE:

```
# Tic Tac Toe Solver using Minimax Algorithm

# Constants for the players
PLAYER_X = 'X'
PLAYER_O = 'O'
EMPTY = ' '

# Function to print the current board state
def print_board(board):
    """Prints the Tic Tac Toe board in a readable format."""
    for row in range(3):
        print("|".join(board[row]))
        if row < 2:
            print("-" * 5)
    print()

# Function to check if the game is over (win or draw)
def is_game_over(board):
    """Returns True if the game is over (win or draw), False otherwise."""
    # Check for winning condition
    for row in range(3):
        if board[row][0] == board[row][1] == board[row][2] != EMPTY:  # Check rows
            return True
    for col in range(3):
        if board[0][col] == board[1][col] == board[2][col] != EMPTY:  # Check columns
            return True
    if board[0][0] == board[1][1] == board[2][2] != EMPTY:  # Check diagonals
        return True
    if board[0][2] == board[1][1] == board[2][0] != EMPTY:
        return True

    # Check for a draw (if no empty spaces remain)
    for row in range(3):
```

```
        for col in range(3):
            if board[row][col] == EMPTY:
                return False  # Game is not over yet, as there are empty spots
    return True  # It's a draw if no empty spots

# Function to evaluate the score of the current board
def evaluate(board):
    """Evaluates the board and returns a score."""
    # Check rows, columns, and diagonals for a winner
    for row in range(3):
        if board[row][0] == board[row][1] == board[row][2]:
            if board[row][0] == PLAYER_X:
                return 10
            elif board[row][0] == PLAYER_O:
                return -10

    for col in range(3):
        if board[0][col] == board[1][col] == board[2][col]:
            if board[0][col] == PLAYER_X:
                return 10
            elif board[0][col] == PLAYER_O:
                return -10

    if board[0][0] == board[1][1] == board[2][2]:
        if board[0][0] == PLAYER_X:
            return 10
        elif board[0][0] == PLAYER_O:
            return -10

    if board[0][2] == board[1][1] == board[2][0]:
        if board[0][2] == PLAYER_X:
            return 10
        elif board[0][2] == PLAYER_O:
            return -10

    return 0  # No winner yet, return 0

# Minimax function to determine the best move
def minimax(board, depth, is_maximizing):
    """
    Minimax algorithm to choose the best possible move.
    :param board: current state of the board
    :param depth: current depth of the recursion (how many moves ahead)
    :param is_maximizing: True if the current player is maximizing (X), False if minimizing
(O)
    :return: the score of the board after evaluating
    """
    score = evaluate(board)

    # If the game is over, return the score
    if score == 10 or score == -10:
        return score
```

```
    if is_game_over(board):
        return 0

    if is_maximizing:  # Maximize for PLAYER_X
        best = -float('inf')
        for row in range(3):
            for col in range(3):
                if board[row][col] == EMPTY:
                    board[row][col] = PLAYER_X  # Make the move
                    best = max(best, minimax(board, depth + 1, not is_maximizing))
                    board[row][col] = EMPTY  # Undo the move
        return best
    else:  # Minimize for PLAYER_O
        best = float('inf')
        for row in range(3):
            for col in range(3):
                if board[row][col] == EMPTY:
                    board[row][col] = PLAYER_O  # Make the move
                    best = min(best, minimax(board, depth + 1, not is_maximizing))
                    board[row][col] = EMPTY  # Undo the move
        return best

# Function to find the best move for the current player
def find_best_move(board):
    """
    Finds the best move for the current player (PLAYER_X).
    :param board: current state of the board
    :return: the best move as a tuple (row, col)
    """
    best_val = -float('inf')
    best_move = (-1, -1)

    # Loop through all empty spots to evaluate the best move
    for row in range(3):
        for col in range(3):
            if board[row][col] == EMPTY:
                board[row][col] = PLAYER_X  # Make the move
                move_val = minimax(board, 0, False)  # Call minimax for minimizing player (O)
                board[row][col] = EMPTY  # Undo the move

                if move_val > best_val:
                    best_move = (row, col)
                    best_val = move_val

    return best_move

# Main function to run the game
def play_game():
    """Function to play the game, showing the board and the moves."""
    board = [[EMPTY for _ in range(3)] for _ in range(3)]  # Initialize an empty board
```

```python
    while not is_game_over(board):
        print_board(board)

        # Player X's turn
        print("Player X's turn:")
        row, col = find_best_move(board)
        board[row][col] = PLAYER_X

        if is_game_over(board):
            print_board(board)
            print("Player X wins!")
            return

        print_board(board)

        # Player O's turn (Human or AI)
        print("Player O's turn (enter row and col):")
        try:
            row, col = map(int, input().split())
            if board[row][col] != EMPTY:
                print("Invalid move, try again.")
                continue
            board[row][col] = PLAYER_O
        except (ValueError, IndexError):
            print("Invalid input. Please enter two numbers between 0 and 2.")
            continue

    print_board(board)
    print("It's a draw!")

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

```
 | |
-----
 | |
-----
 | |

Player X's turn:
X| |
-----
 | |
-----
 | |

Player O's turn (enter row and col):
1 2
X| |
-----
 | |O
-----
 | |

Player X's turn:
X| |X
-----
 | |O
-----
 | |

Player O's turn (enter row and col):
2 2
X| |X
-----
 | |O
-----
 | |O

Player X's turn:
X|X|X
-----
 | |O
-----
 | |O

Player X wins!
```

OUTPUT :