Introduction

Tic-Tac-Toe, also known as Noughts and Crosses or Xs and Os, is a popular two-player game where players take turns marking spaces in a 3x3 grid. The goal of the game is to be the first to get three of their marks in a horizontal, vertical, or diagonal line. The game is commonly used as a pastime and is often used as an introductory example for various programming techniques and algorithms.

In this report, we explore a **Tic-Tac-Toe Solver**, which uses the **Minimax Algorithm** to evaluate the best moves for a player. The solver aims to simulate the best possible moves for a player playing as "X" (AI), against an opponent playing as "O" (human or AI). The Minimax algorithm is a decision-making algorithm that is often used in games involving two players and zero-sum situations, where one player's gain is the other's loss.

The solver presented here uses recursion to simulate all possible game states and evaluates the best move to maximize the chances of winning. This report provides an overview of the problem-solving process, the methodology used, the code implementation, and the results of the Tic-Tac-Toe Solver.

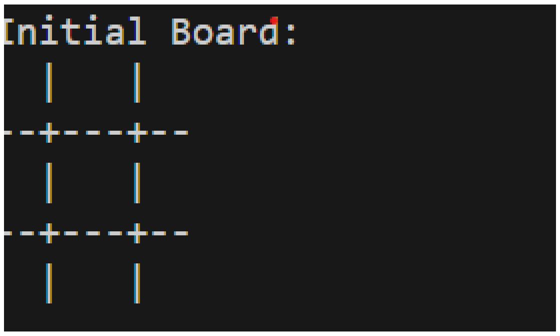
Methodology

The methodology for implementing the Tic-Tac-Toe Solver involves the following steps:

1. **Game Representation**: The game board is represented as a 1D list of 9 elements (a 3x3 grid). Each element can be 'X', 'O', or a blank space (' ') to indicate the state of the cell.
2. **Game Evaluation**:
    - **Check for Win**: A function is implemented to determine if a given player ('X' or 'O') has won the game based on the positions of their marks on the grid.
    - **Check for Draw**: A function checks if the board is full and there are no winners, resulting in a draw.
3. **Minimax Algorithm**:
    - **Recursion**: The algorithm uses recursion to simulate each possible move by alternating between maximizing (AI's move) and minimizing (opponent's move).
    - **Base Case**: The algorithm stops when a terminal state is reached, i.e., either a player wins or the game ends in a draw.
    - **Evaluation Function**: Each board state is evaluated based on the current depth in the recursion. If the AI wins, the score is positive; if the opponent wins, the score is negative; if the game is a draw, the score is 0.
    - **Optimal Move**: The algorithm searches through all possible moves and chooses the one that leads to the best outcome (either winning or minimizing the loss).
4. **Implementation**: Python was chosen to implement the algorithm because of its simplicity and the availability of libraries that facilitate recursive and list operations.

Code Implementation

```
import math


# Initialize the board
board = [' ' for _ in range(9)]  # 3x3 Tic-Tac-Toe grid
```



```
Initial Board:
  |   |
--+---+--
  |   |
--+---+--
  |   |
```

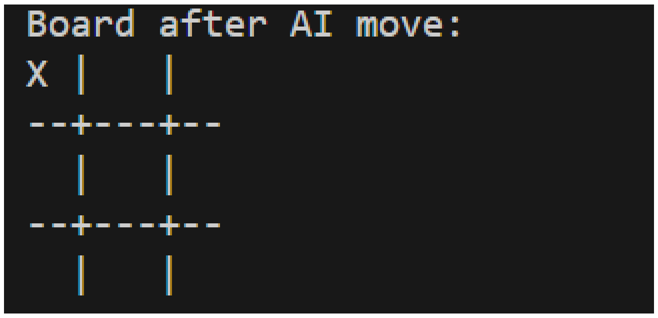
```
# Check if a player has won
def check_win(board, player):
        win_positions = [(0, 1, 2), (3, 4, 5), (6, 7, 8),  # Rows
                (0, 3, 6), (1, 4, 7), (2, 5, 8),  # Columns
                (0, 4, 8), (2, 4, 6)]          # Diagonals
        return any(board[a] == board[b] == board[c] == player for a, b, c in win_positions)


# Check if the board is full
def is_full(board):
        return ' ' not in board


# Minimax Algorithm
def minimax(board, depth, is_maximizing):
```

```
if check_win(board, 'X'):

return 10 - depth

if check_win(board, 'O'):

return depth - 10

if is_full(board):

return 0


if is_maximizing:

best = -math.inf

for i in range(9):

if board[i] == ' ':

   board[i] = 'X'

        best = max(best, minimax(board, depth + 1, False))

   board[i] = ' '

return best

else:

best = math.inf

for i in range(9):

if board[i] == ' ':

   board[i] = 'O'

        best = min(best, minimax(board, depth + 1, True))

   board[i] = ' '

return best
```

```
# Find the best move for the AI (X)

def find_best_move(board):

    best_value = -math.inf

    best_move = -1


    for i in range(9):

    if board[i] == ' ':

    board[i] = 'X'

    move_value = minimax(board, 0, False)

    board[i] = ' '

    if move_value > best_value:

        best_value = move_value

        best_move = i


    return best_move


# Display the board

def print_board(board):

    for i in range(0, 9, 3):

  print(board[i], '|', board[i+1], '|', board[i+2])

    if i < 6:

    print('--+---+--')

    print()
```

```
# Example of running the solver

print("Initial Board:")

print_board(board)


move = find_best_move(board)

print(f"AI's best move is at index {move}")


# Make the AI move

board[move] = 'X'

print("Board after AI move:")

print_board(board)
```

output



```
Board after AI move:
X |   |
--+---+--
  |   |
--+---+--
  |   |
```