# Tic Tac Toe using MinMax Algorithm

**Aim:**

To implement and evaluate the performance of the Minimax algorithm for playing the Tic Tac Toe game, where the AI (Player O) makes optimal moves based on the evaluation of all possible game states, and to test the effectiveness of the algorithm in winning, drawing, or losing against a human player (Player X).

**Description:**

Tic Tac Toe is a two-player board game played on a 3x3 grid. The goal is to place three of one's marks (either X or O) in a row, column, or diagonal before the opponent. In this problem, the objective is to implement an AI (Player O) that can play optimally using the Minimax algorithm against a human player (Player X). The Minimax algorithm is a decision-making process used to determine the best move by evaluating all possible moves in the game. It simulates each possible move, recursively calculates the resulting game state, and assigns a score to each state based on whether the AI wins, loses, or draws. The algorithm then selects the move that maximizes the AI's chances of winning, assuming the opponent plays optimally. The AI's decisions are driven by these evaluations, making it capable of competing at an optimal level with the player. The problem involves implementing the game logic, including the board setup, player turns, checking for a win or draw, and using the Minimax algorithm to choose the best possible move for the AI. The effectiveness of the AI can be tested through various game scenarios, where it either wins, draws, or loses against a human player.

**Code:**

```
#include <iostream>

#include <vector>


using namespace std;


const char PLAYER_X = 'X';

const char PLAYER_O = 'O';

const char EMPTY = ' ';


// Function to print the board

void printBoard(const vector<vector<char>>& board) {

    cout << "-------------\n";

    for (int i = 0; i < 3; i++) {
```

```
        cout << "| ";
        for (int j = 0; j < 3; j++) {
            cout << board[i][j] << " | ";
        }
        cout << endl;
        cout << "-------------\n";
    }
}

// Function to check if the current player has won
bool isWinning(const vector<vector<char>>& board, char player) {
    // Check rows and columns
    for (int i = 0; i < 3; i++) {
        if ((board[i][0] == player && board[i][1] == player && board[i][2] == player) ||
            (board[0][i] == player && board[1][i] == player && board[2][i] == player)) {
            return true;
        }
    }
    // Check diagonals
    if ((board[0][0] == player && board[1][1] == player && board[2][2] == player) ||
        (board[0][2] == player && board[1][1] == player && board[2][0] == player)) {
        return true;
    }
    return false;
}

// Function to check if the game is a draw
bool isDraw(const vector<vector<char>>& board) {
    for (int i = 0; i < 3; i++) {
```

```
        for (int j = 0; j < 3; j++) {
            if (board[i][j] == EMPTY) {
                return false;
            }
        }
    }
    return true;
}

// Minimax algorithm to calculate the best move for the AI
int minimax(vector<vector<char>>& board, int depth, bool isMaximizingPlayer) {
    if (isWinning(board, PLAYER_X)) return -10 + depth;
    if (isWinning(board, PLAYER_O)) return 10 - depth;
    if (isDraw(board)) return 0;

    if (isMaximizingPlayer) {
        int best = -1000;
        for (int i = 0; i < 3; i++) {
            for (int j = 0; j < 3; j++) {
                if (board[i][j] == EMPTY) {
                    board[i][j] = PLAYER_O;
                    best = max(best, minimax(board, depth + 1, false));
                    board[i][j] = EMPTY;
                }
            }
        }
        return best;
    } else {
        int best = 1000;
```

```
        for (int i = 0; i < 3; i++) {
            for (int j = 0; j < 3; j++) {
                if (board[i][j] == EMPTY) {
                    board[i][j] = PLAYER_X;
                    best = min(best, minimax(board, depth + 1, true));
                    board[i][j] = EMPTY;
                }
            }
        }
        return best;
    }
}

// Function to find the best move for the AI (Player O)
pair<int, int> findBestMove(vector<vector<char>>& board) {
    int bestVal = -1000;
    pair<int, int> bestMove = {-1, -1};

    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            if (board[i][j] == EMPTY) {
                board[i][j] = PLAYER_O;
                int moveVal = minimax(board, 0, false);
                board[i][j] = EMPTY;
                if (moveVal > bestVal) {
                    bestMove = {i, j};
                    bestVal = moveVal;
                }
            }
```

```
        }
    }
    return bestMove;
}

// Function to play the game
void playGame() {
    vector<vector<char>> board(3, vector<char>(3, EMPTY));
    int row, col;
    bool playerTurn = true; // true means Player X's turn, false means Player O's turn

    while (true) {
        printBoard(board);

        if (playerTurn) {
            cout << "Player X's turn. Enter row and column (0-2): ";
            cin >> row >> col;
            if (row < 0 || row >= 3 || col < 0 || col >= 3 || board[row][col] != EMPTY) {
                cout << "Invalid move, try again.\n";
                continue;
            }
            board[row][col] = PLAYER_X;
        } else {
            cout << "Player O's turn (AI). Computing best move...\n";
            pair<int, int> bestMove = findBestMove(board);
            board[bestMove.first][bestMove.second] = PLAYER_O;
        }

        if (isWinning(board, PLAYER_X)) {
```

```
            printBoard(board);
            cout << "Player X wins!\n";
            break;
        }
        if (isWinning(board, PLAYER_O)) {
            printBoard(board);
            cout << "Player O wins!\n";
            break;
        }
        if (isDraw(board)) {
            printBoard(board);
            cout << "It's a draw!\n";
            break;
        }

        playerTurn = !playerTurn; // Switch turns
    }
}

int main() {
    playGame();
    return 0;
}
```

**Output:**

```
-------------
|   |   |   |
-------------
|   |   |   |
-------------
|   |   |   |
-------------
Player X's turn. Enter row and column (0-2): 0 0
-------------
| X |   |   |
-------------
|   |   |   |
-------------
|   |   |   |
-------------
```

```
Player O's turn (AI). Computing best move...
-------------
| X |   |   |
-------------
|   | O |   |
-------------
|   |   |   |
-------------
Player X's turn. Enter row and column (0-2): 0 2
-------------
| X |   | X |
-------------
|   | O |   |
-------------
|   |   |   |
-------------
```

```
-------------
Player O's turn (AI). Computing best move...
-------------
| X | O | X |
-------------
|   | O |   |
-------------
|   |   |   |
-------------
Player X's turn. Enter row and column (0-2): 1 2
-------------
| X | O | X |
-------------
|   | O | X |
-------------
|   |   |   |
-------------
```

```
-------------
Player O's turn (AI). Computing best move...
-------------
| X | O | X |
-------------
|   | O | X |
-------------
|   | O |   |
-------------
Player O wins!
```

**Code Explanation:**

1. **Board Representation:**
    - The board is represented as a 2D vector of characters ('X', 'O', and ' ' for empty spaces). It's a 3x3 grid used for storing the game state.
2. **Utility Functions:**
    - printBoard(): Displays the current state of the game board to the console.
    - isWinning(): Checks if a given player ('X' or 'O') has won the game by checking rows, columns, and diagonals.
    - isDraw(): Checks if the game has resulted in a draw, i.e., no more moves are available and no player has won.

3. **Minimax Algorithm:**
    - The core of the AI's decision-making process is the minimax() function. It recursively evaluates all possible moves for the AI and the player, assigning scores based on the outcome of each potential game state:
        - Positive scores for AI wins (Player O).
        - Negative scores for player wins (Player X).
        - Zero for a draw.
    - The minimax() function works by simulating every possible move and evaluating it. The AI (maximizing player) tries to maximize the score, while the player (minimizing player) tries to minimize the score.
4. **Finding the Best Move for AI:**
    - The findBestMove() function determines the best possible move for the AI by simulating all moves and choosing the one with the highest score, calculated by the Minimax algorithm.
5. **Game Loop:**
    - The playGame() function is the main loop of the game, alternating between Player X (human) and Player O (AI). It prompts the human player to enter a move, checks for a win or draw after every move, and calls the minimax() function when it's the AI's turn to calculate the best possible move.
    - The game ends when either a player wins or the board is full, resulting in a draw.
6. **Main Function:**
    - The main() function simply calls playGame() to start the game.

**Time Complexity:**

1. **Game Tree Depth**: The maximum depth of the game tree is 9 (since the board has 9 positions).
2. **Branching Factor**: At each level of the tree, the number of possible moves decreases. For the first move, there are 9 options, then 8 for the next, and so on.
3. **Total Number of States**: The total number of possible game states is the number of ways to arrange the 9 positions, which is **9! (factorial of 9)**.

**Space Complexity:**

1. The space complexity of the Minimax algorithm is **O(d)**, where d is the depth of the recursion (9 in this case).
2. Since d is a constant for Tic Tac Toe, the space complexity can be considered **O(1)** in practice for this specific problem, as the recursion depth and board size are fixed.