File :

🧪 Experiment Number: 1

📘 Subject: Artificial Intelligence

📚 Semester: 5th Semester

🎯 Aim

To write a program to implement the **Tic Tac Toe** game using the **Minimax algorithm** for AI decision-making.

🧠 Theory

Tic Tac Toe Overview

Tic Tac Toe is a two-player game played on a 3×3 grid where players take turns marking spaces with "X" or "O". The first player to align three of their marks horizontally, vertically, or diagonally wins.

Minimax Algorithm

The **Minimax algorithm** is a decision-making algorithm used in two-player turn-based games like Tic Tac Toe. It assumes both players play optimally:

- The **Maximizing player** (AI) tries to **maximize** the score.
- The **Minimizing player** (human) tries to **minimize** the score.

Scoring Logic:

- Win: +10
- Loss: -10
- Draw: 0

Working Principle:

1. Simulate all possible moves.

2. Recursively evaluate each game state.
3. Choose the move with the best evaluated outcome.

🧮 Algorithm (Minimax)

1. Check if the game is over (win/lose/draw).
2. If it's AI's turn (Maximizer):
   - Set `best = -∞`
   - For each valid move:
     - Make the move.
     - Call `minimax()` recursively for the opponent.
     - Undo the move.
     - Update `best = max(best, returned value)`
   - Return `best`
3. If it's the human's turn (Minimizer):
   - Set `best = +∞`
   - For each valid move:
     - Make the move.
     - Call `minimax()` recursively for AI.
     - Undo the move.
     - Update `best = min(best, returned value)`
   - Return `best`

💻 Code (C++ Implementation)

```cpp
#include <iostream>
#include <limits>
using namespace std;

const char PLAYER = 'O';
const char AI = 'X';

char board[3][3] = {
    { '1', '2', '3' },
    { '4', '5', '6' },
    { '7', '8', '9' }
};

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

```
        for (int j = 0; j < 3; j++)
            if (board[i][j] != 'X' && board[i][j] != 'O')
                return true;
    return false;
}

int evaluate() {
    // Check rows, columns, and diagonals
    for (int row = 0; row < 3; row++)
        if (board[row][0] == board[row][1] &&
            board[row][1] == board[row][2])
            return (board[row][0] == AI) ? +10 : -10;

    for (int col = 0; col < 3; col++)
        if (board[0][col] == board[1][col] &&
            board[1][col] == board[2][col])
            return (board[0][col] == AI) ? +10 : -10;

    if (board[0][0] == board[1][1] &&
        board[1][1] == board[2][2])
        return (board[0][0] == AI) ? +10 : -10;

    if (board[0][2] == board[1][1] &&
        board[1][1] == board[2][0])
        return (board[0][2] == AI) ? +10 : -10;

    return 0;
}

int minimax(bool isMax) {
    int score = evaluate();
    if (score == 10 || score == -10)
        return score;
    if (!isMovesLeft())
        return 0;

    if (isMax) {
        int best = -1000;
        for (int i = 0; i < 3; i++)
            for (int j = 0; j < 3; j++)
                if (board[i][j] != 'X' && board[i][j] != 'O') {
                    char backup = board[i][j];
                    board[i][j] = AI;
                    best = max(best, minimax(false));
                    board[i][j] = backup;
                }
        return best;
    } else {
        int best = 1000;
        for (int i = 0; i < 3; i++)
```

```
            for (int j = 0; j < 3; j++)
                if (board[i][j] != 'X' && board[i][j] != 'O') {
                    char backup = board[i][j];
                    board[i][j] = PLAYER;
                    best = min(best, minimax(true));
                    board[i][j] = backup;
                }
        return best;
    }
}

pair<int, int> findBestMove() {
    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] != 'X' && board[i][j] != 'O') {
                char backup = board[i][j];
                board[i][j] = AI;
                int moveVal = minimax(false);
                board[i][j] = backup;

                if (moveVal > bestVal) {
                    bestMove = {i, j};
                    bestVal = moveVal;
                }
            }
    return bestMove;
}
```

🔢 Inputs/Outputs Example

Input:

- A partially filled board where AI and player take turns.
- Human inputs a number (1–9) to make a move.

Output:

- Updated board after each move
- AI calculates and plays optimally
- Final result: Win / Lose / Draw message

📈 Analysis of Code and Algorithm

- **Minimax** explores the entire game tree recursively.
- **evaluate()** checks if there's a winner and assigns score.
- **findBestMove()** chooses the move with the highest score for AI.
- Deterministic: the AI always picks the best possible move.

Complexities:

- **Time Complexity**: O(9!) in worst-case (full game tree traversal)
- **Space Complexity**: O(1) (excluding recursion stack)

🌐 Real-Life Applications

- Game AI development (Chess, Checkers, etc.)
- Turn-based decision-making systems
- Teaching game theory, AI search, and recursion fundamentals

---

✅ Conclusion

This experiment successfully implemented a classic **Tic Tac Toe** game using the **Minimax algorithm**, allowing the AI to make optimal moves. We explored how AI can simulate human decision-making and used recursive strategies to evaluate every possible outcome. This experiment enhances understanding of adversarial search, game theory, and intelligent agent behavior.