CSE-AI TY A div

Student Name: Prachee Prasad

Roll No.: 381060

Assignment 5

Implement Minimax Algorithm for Game Playing

Problem Statement:

To implement the **Minimax Algorithm** for a simple two-player game, enabling the computer to make optimal decisions by minimizing possible losses in a worst-case scenario.

Objective:

To understand how **decision-making** in Artificial Intelligence works using the **Minimax Algorithm**, which simulates possible moves in a game and chooses the best one assuming the opponent also plays optimally.

Requirements:

- Input: Predefined game tree or static evaluation function (e.g., Tic-Tac-Toe, simplified numeric game).
- Output: Step-by-step evaluation of moves and the optimal decision chosen by the AI player.
- Approach: Recursive Minimax traversal evaluating possible outcomes.

Operating System:

Windows / Linux / macOS

Libraries and Packages Used:

- **C++ iostream**, **algorithm**, and **vector** for recursive computation and data handling.

- No external libraries required.

Theory:

Definition:

The **Minimax Algorithm** is a recursive search algorithm used in **game theory** and **AI** for decision-making in **two-player zero-sum games**.
It assumes that both players play optimally—one tries to **maximize** the score, and the other tries to **minimize** it.

Structure:

- **Maximizer:** The player who tries to achieve the highest possible score.
- **Minimizer:** The opponent who tries to minimize the maximizer's score.
- **Game Tree:** Represents all possible game states as nodes.
- **Terminal Nodes:** Represent end states with an evaluation value (win/loss/draw).

Methodology:

1. Represent the game as a **tree structure**, where each node corresponds to a game state.
2. The algorithm recursively explores all possible future moves.
3. If it's the maximizer's turn → choose the **maximum** value among the child nodes.
4. If it's the minimizer's turn → choose the **minimum** value among the child nodes.
5. Continue until reaching terminal nodes (end of game).
6. Backtrack to the root node to find the **optimal move**.

Advantages:

- Ensures **optimal gameplay** under perfect play conditions.
- Provides a **logical and systematic** way of decision-making.

- Forms the basis for more advanced algorithms like **Alpha-Beta Pruning**.

Limitations:

- **Computationally expensive** for large game trees.
- Requires **complete knowledge** of possible moves.
- Doesn't handle uncertainty or randomness well (requires deterministic games).

Working / Algorithm:

Algorithm Steps:

1. Start from the **root node** (current game state).
2. Generate all possible **successor states** (moves).
3. Recursively apply Minimax to each successor:
    - If **MAX's turn**, return the **maximum** of child scores.
    - If **MIN's turn**, return the **minimum** of child scores.
4. Continue recursion until reaching a **terminal node** with a static score.
5. Backtrack values to the root to find the **best move** for MAX (AI).

Example Working:

```
Game Tree Example:
           (?)
         /  |  \
        3   5   2
Player (MAX) chooses the highest value → 5
```

Hence, the **optimal move** selected by the AI is **5**.

Conclusion:

The **Minimax Algorithm** enables computers to make strategic decisions by evaluating all possible moves and counter-moves.
It forms the backbone of **AI-based game engines**, providing an intelligent foundation for adversarial problem-solving in games like **Chess**, **Tic-Tac-Toe**, and **Checkers**.