**Assignment 5**

**Title:**

Implementation of the Minimax Algorithm for Two-Player Game Playing

**Aim:**

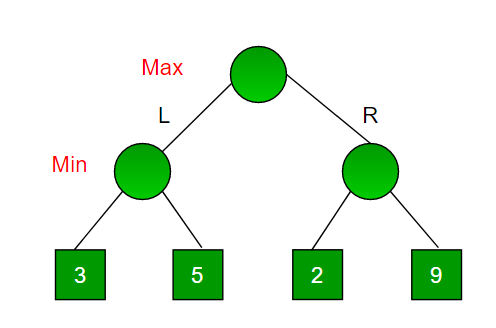
To implement the **Minimax Algorithm** for a two-player game like Tic-Tac-Toe and evaluate its decision-making process to determine optimal moves for both players.

**Objectives:**

1. To understand the **Minimax Algorithm** and its role in decision-making for two-player zero-sum games.
2. To implement the Minimax Algorithm to find optimal moves for a given game state.
3. To simulate a two-player game (e.g., Tic-Tac-Toe) where one or both players use the Minimax algorithm for decision-making.
4. To evaluate the algorithm's ability to ensure that each player plays optimally, minimizing the opponent's maximum payoff.

**Theory:**

**Minimax Algorithm Overview:**



The **Minimax algorithm** is a recursive decision-making algorithm used in two-player games like Tic-Tac-Toe, Chess, and Checkers. The algorithm is designed for **zero-sum games**, where one player’s gain is equivalent to the other player’s loss. It assumes that both players play optimally, meaning they make the best moves possible at each turn.

* **Maximizing Player (MAX)**: This player tries to **maximize** their score.
* **Minimizing Player (MIN)**: This player tries to **minimize** the maximizing player's score.

The algorithm works by simulating all possible moves, considering both players' actions, and choosing the move that maximizes the score for MAX while minimizing the possible gain of the opponent (MIN). It evaluates game states using a **heuristic or evaluation function**, typically applied to end-game conditions (win/loss/draw).

**Algorithm Steps:**

1. **Generate Game Tree**:
   * At each game state, generate all possible moves for the current player.
2. **Evaluate Terminal Nodes**:
   * If a game state is a terminal state (win/loss/draw), assign a value (e.g., +1 for win, -1 for loss, 0 for draw).
3. **Minimax Decision**:
   * Propagate the values from terminal nodes up the tree by having the MAX player choose the maximum value from possible moves and the MIN player choose the minimum value.
4. **Backtracking**:
   * Recursively backtrack until the root node is reached, selecting the optimal move for MAX or MIN at each level.
5. **Pruning (Optional)**:
   * Use techniques like **Alpha-Beta pruning** to eliminate unnecessary branches from the game tree for faster computation.

**Procedure:**

**1. Define the Game (Tic-Tac-Toe Example):**

* The game is played on a **3x3 grid**, with two players alternating turns.
* Player 1 uses X, and Player 2 uses O.
* The goal is to align 3 of your marks (either X or O) in a row, column, or diagonal.
* The game ends with a win, loss, or draw.

**2. Define the Evaluation Function:**

* For Tic-Tac-Toe, the evaluation function assigns values to terminal game states:
  + **+1** if Player 1 (MAX) wins.
  + **-1** if Player 2 (MIN) wins.
  + **0** for a draw.

**3. Implement the Minimax Algorithm:**

* **Recursive Function**:
  + The algorithm is implemented as a recursive function where:
    1. If the current state is a terminal state (win/loss/draw), return the evaluation value.
    2. If it is the maximizing player's turn (X), explore all possible moves and return the move that gives the maximum value.
    3. If it is the minimizing player's turn (O), explore all possible moves and return the move that gives the minimum value.
* **Game Tree Exploration**:
  + At each level of the game tree, simulate all possible moves and calculate their outcomes recursively.

**4. Simulate the Game:**

* Alternate turns between Player 1 (X) and Player 2 (O).
* For Player 1 (X), use the Minimax algorithm to make optimal moves.
* For Player 2 (O), you can either:
  + Use the Minimax algorithm for optimal moves (two AI agents).
  + Allow a human player to play against the AI.

**5. Game Termination:**

* The game ends when:
  1. One player wins by aligning three of their marks in a row, column, or diagonal.
  2. The grid is full, resulting in a draw.

**6. Implement Alpha-Beta Pruning (Optional):**

* Optimize the Minimax algorithm by pruning branches that cannot affect the final decision.
* **Alpha**: The best value that the maximizer currently can guarantee.
* **Beta**: The best value that the minimizer currently can guarantee.
* This reduces the number of nodes explored in the game tree, speeding up computation.

**Expected Output:**

1. **Game Simulation**:
   * A step-by-step simulation of Tic-Tac-Toe where the AI (using Minimax) makes optimal moves.
   * If Player 1 is using the Minimax algorithm, it should always play optimally, trying to win or force a draw.
2. **Optimal Moves**:
   * For each turn, the algorithm will output the optimal move for Player 1 (X), and optionally for Player 2 (O) if both players are AI-controlled.
3. **Game Outcome**:
   * The game should either end with Player 1 (X) winning, Player 2 (O) winning, or a draw.

**Procedure for Example Problem (Tic-Tac-Toe):**

1. **Define the Game Board**:
   * Initialize a 3x3 grid with empty spaces.
   * Example:

Board:

\_ \_ \_

\_ \_ \_

\_ \_ \_

1. **Player Turns**:

* Alternate turns between Player 1 (X) and Player 2 (O).
* After each move, check if the game has ended (win/loss/draw).

1. **Minimax Algorithm**:

* For Player 1 (X), use the Minimax algorithm to calculate the best possible move.
* For Player 2 (O), either allow human input or use the Minimax algorithm for automated play.

1. **Optimal Play**:

* Example moves:

Board after few moves:

X O X

\_ X \_

O \_ \_

* The Minimax algorithm calculates the optimal move for Player 1 (X) to win or force a draw.

1. **Game Termination**:

* The game ends when either player wins or all cells are filled, resulting in a draw.

**Conclusion:**

In this lab, the **Minimax algorithm** was successfully implemented to play a two-player game (Tic-Tac-Toe). The algorithm enabled Player 1 to make optimal decisions by minimizing the potential gain of the opponent (Player 2). The algorithm ensures that the player using it never loses, and always either wins or forces a draw. Additionally, the optional implementation of **Alpha-Beta pruning** allowed for an optimized solution by reducing the number of game tree nodes explored.