**Lab Assignment 5: Implementation of Minimax Algorithm for Game Playing**

**1. Objective:**

The objective of this lab assignment is to implement the **Minimax algorithm** for decision-making in a two-player, turn-based game. The Minimax algorithm will enable one of the players (typically the AI) to make optimal moves by evaluating all possible future moves, assuming the opponent plays optimally as well.

**2. Problem Statement:**

Implement the **Minimax algorithm** for a simple turn-based game, such as **Tic-Tac-Toe** or **Connect Four**. The algorithm should evaluate all possible moves, assess the resulting game states, and select the best move based on the principle of minimizing the opponent’s chances of winning while maximizing its own chances.

**3. Theory:**

**3.1. Minimax Algorithm Overview:**

The **Minimax algorithm** is used in two-player games to determine the best possible move by simulating all future game states. It assumes that both players play optimally. Minimax operates by generating a **game tree**, where each node represents a game state, and the edges represent possible moves.

* **Maximizer**: The player trying to maximize their score (typically the AI).
* **Minimizer**: The opponent trying to minimize the AI’s score.

Minimax evaluates the game tree by assigning values to the terminal nodes (game-ending states, such as win, lose, or draw) and propagating these values back up the tree to determine the best move.

**3.2. Minimax Function:**

The Minimax algorithm works recursively:

* If the current player is the maximizer, the algorithm chooses the move that **maximizes** the value of the resulting game state.
* If the current player is the minimizer, the algorithm chooses the move that **minimizes** the value of the resulting game state.

**3.3. Game Evaluation Function:**

An **evaluation function** assigns a score to each game state:

* **Win for Maximizer**: +10.
* **Win for Minimizer**: -10.
* **Draw**: 0.

For example, in **Tic-Tac-Toe**, the evaluation function might return:

* +10 if the AI wins.
* -10 if the opponent wins.
* 0 for a draw.

**4. Algorithm Design:**

**4.1. Minimax Algorithm Steps:**

1. **Generate Game Tree**:
   * Create a tree where each node represents a game state, and the children of a node represent all possible moves from that state.
2. **Assign Values to Terminal Nodes**:
   * For each terminal node (end-game state), assign a score based on whether it’s a win, loss, or draw.
3. **Propagate Values Up the Tree**:
   * For each non-terminal node, determine the best move by choosing the maximum score (if it's the AI’s turn) or the minimum score (if it’s the opponent’s turn).
4. **Make Optimal Move**:
   * At the root of the tree (current game state), select the move corresponding to the best propagated score.

**4.2. Pseudocode:**

Here is the pseudocode for the Minimax algorithm in a two-player game like Tic-Tac-Toe:

php

Copy code

function minimax(board, depth, isMaximizingPlayer):

if terminal\_state(board):

return evaluate(board)

if isMaximizingPlayer:

best = -∞

for each move in possible\_moves(board):

make\_move(board, move)

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

undo\_move(board, move)

return best

else:

best = +∞

for each move in possible\_moves(board):

make\_move(board, move)

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

undo\_move(board, move)

return best

function best\_move(board):

best\_value = -∞

best\_move = None

for each move in possible\_moves(board):

make\_move(board, move)

move\_value = minimax(board, 0, false)

undo\_move(board, move)

if move\_value > best\_value:

best\_value = move\_value

best\_move = move

return best\_move

**4.3. Explanation of the Code:**

* **minimax(board, depth, isMaximizingPlayer)**: This function recursively evaluates the board. It checks all possible moves, calculates the resulting scores for the game states, and chooses the maximum score for the maximizer and the minimum score for the minimizer.
* **terminal\_state(board)**: This function checks if the current board is a terminal state (win, loss, or draw).
* **evaluate(board)**: This function returns a value for the current board state based on the outcome (win, loss, or draw).
* **possible\_moves(board)**: Returns a list of all legal moves that can be made on the current board.
* **best\_move(board)**: This function selects the move that results in the highest Minimax score for the AI.

**5. Application: Minimax in Tic-Tac-Toe**

**5.1. Game Rules:**

* Tic-Tac-Toe is a two-player game played on a 3x3 grid.
* Players take turns marking a cell with their symbol (X or O).
* The game ends when one player gets three of their symbols in a row, column, or diagonal, or when the grid is full (draw).

**5.2. Board Representation:**

The board can be represented as a 2D array:

css

Copy code

[ ['X', 'O', 'X'],

['O', 'X', ' '],

[' ', ' ', 'O']

]

**5.3. Evaluation Function:**

The evaluation function in Tic-Tac-Toe might be:

kotlin

Copy code

function evaluate(board):

if AI wins: return +10

if Opponent wins: return -10

return 0 for draw

**5.4. Example Execution:**

Given the current state of the Tic-Tac-Toe board:

css

Copy code

[ ['X', 'O', 'X'],

['O', 'X', ' '],

[' ', ' ', 'O']

]

If it's the AI's turn, the Minimax algorithm will explore all possible moves, evaluate the resulting board states, and choose the move that maximizes the AI's chances of winning (or minimizes the opponent's chances).

**6. Expected Output:**

For the given game board, the algorithm should output the best move for the AI. If the AI is "X" and it’s its turn to play, the algorithm might suggest a move like this:

rust

Copy code

Best move for AI: (2, 2)

The updated board after the AI's move:

css

Copy code

[ ['X', 'O', 'X'],

['O', 'X', ' '],

[' ', ' ', 'X']

]

**7. Optimization: Alpha-Beta Pruning:**

To improve efficiency, you can implement **Alpha-Beta Pruning**, which reduces the number of nodes evaluated by the Minimax algorithm by "pruning" branches that won't affect the final decision.

**Alpha-Beta Pruning Pseudocode:**

php

Copy code

function minimax(board, depth, alpha, beta, isMaximizingPlayer):

if terminal\_state(board):

return evaluate(board)

if isMaximizingPlayer:

maxEval = -∞

for each move in possible\_moves(board):

make\_move(board, move)

eval = minimax(board, depth + 1, alpha, beta, false)

undo\_move(board, move)

maxEval = max(maxEval, eval)

alpha = max(alpha, eval)

if beta <= alpha:

break

return maxEval

else:

minEval = +∞

for each move in possible\_moves(board):

make\_move(board, move)

eval = minimax(board, depth + 1, alpha, beta, true)

undo\_move(board, move)

minEval = min(minEval, eval)

beta = min(beta, eval)

if beta <= alpha:

break

return minEval

**8. Conclusion:**

The Minimax algorithm provides an effective way to simulate and predict outcomes in two-player games like Tic-Tac-Toe. By exploring all possible moves, it ensures that the AI makes the optimal move every time. Alpha-Beta Pruning further optimizes this process by reducing the number of nodes that need to be evaluated.

**9. References:**

* Russell, S. J., & Norvig, P. (2020). *Artificial Intelligence: A Modern Approach* (4th ed.). Pearson.
* Nilsson, N. J. (1998). *Artificial Intelligence: A New Synthesis*. Morgan Kaufmann.