1. Aim

To understand and implement the **Minimax algorithm** for a two-player deterministic, zero-sum game (Tic-Tac-Toe), evaluate its performance, and discuss its limitations and possible improvements.

🎯 2. Learning Objectives

By the end of this lab, the student will be able to:

- Explain the **Minimax decision rule** and **game tree concept**.
- Implement **Minimax using recursion** to choose optimal moves.
- Apply **Minimax to Tic-Tac-Toe** and verify correct, optimal play.
- Analyze **algorithmic complexity** and discuss **Alpha-Beta pruning** and **depth-limiting techniques**.

📘 3. Background / Theory

- **Two-player deterministic games** like Tic-Tac-Toe and Chess can be modeled as **game trees**, where:
    - Each **node** represents a game state.
    - Each **edge** represents a possible move.
    - Players alternate turns — **MAX** tries to maximize utility, **MIN** tries to minimize it.
    - It's a **zero-sum game**, meaning one player's gain is another's loss.

Minimax Concept:

1. Explore all possible future moves up to terminal states (win, loss, draw).
2. Evaluate each terminal state with a **utility function**:
    - Win = +1
    - Draw = 0
    - Loss = –1
3. Propagate these values back up the tree:
    - At **MAX** nodes → choose maximum utility.
    - At **MIN** nodes → choose minimum utility.

Complexity:

- **Time Complexity:** O(b^d), where
  b = branching factor,
  d = depth of the game tree.
- **Tic-Tac-Toe** is small enough to be fully solved using Minimax.
- Larger games require **pruning** (Alpha-Beta) or **heuristic evaluation**.

⚙️ 4. Algorithm (Minimax Pseudocode)

```
minimax(node, depth, player):
    if depth == 0 or node is terminal:
        return value(node)
```

```
    if player == "MAX":
        α = -∞
        for each child of node:
            value = minimax(child, depth-1, "MIN")
            α = max(α, value)
        return α
    else:
        α = +∞
        for each child of node:
            value = minimax(child, depth-1, "MAX")
            α = min(α, value)
        return α
```

💻 5. Python Implementation

```
import math

# Function to check for available moves
def is_moves_left(board):
    return any(cell == ' ' for cell in board)

# Function to evaluate the board
def evaluate(board):
    # Winning combinations
    win_combos = [(0,1,2), (3,4,5), (6,7,8),
                  (0,3,6), (1,4,7), (2,5,8),
                  (0,4,8), (2,4,6)]

    for (x, y, z) in win_combos:
        if board[x] == board[y] == board[z]:
            if board[x] == 'X':  # MAX
                return +1
            elif board[x] == 'O':  # MIN
                return -1
    return 0

# Minimax function
def minimax(board, depth, isMax):
    score = evaluate(board)

    # Terminal state check
    if score == 1:
        return score
    if score == -1:
```

```
        return score
    if not is_moves_left(board):
        return 0

    # Maximizer's move
    if isMax:
        best = -math.inf
        for i in range(9):
            if board[i] == ' ':
                board[i] = 'X'
                best = max(best, minimax(board, depth + 1, False))
                board[i] = ' '
        return best
    # Minimizer's move
    else:
        best = math.inf
        for i in range(9):
            if board[i] == ' ':
                board[i] = 'O'
                best = min(best, minimax(board, depth + 1, True))
                board[i] = ' '
        return best

# Function to find the best move for 'X'
def find_best_move(board):
    best_val = -math.inf
    best_move = -1

    for i in range(9):
        if board[i] == ' ':
            board[i] = 'X'
            move_val = minimax(board, 0, False)
            board[i] = ' '
            if move_val > best_val:
                best_move = i
                best_val = move_val
    return best_move

# Main
board = ['X', 'O', 'X',
         'O', 'O', ' ',
         ' ', 'X', ' ']
```

```
best_move = find_best_move(board)
print ("The optimal move is position:", best_move)
```

🧩 6. Sample Output

The optimal move is position: 5
(Meaning the AI should place 'X' in the 6th cell (index 5) to either win or block opponent.)

📊 7. Observations

Parameter	Observation
Game Type	Two-player, deterministic, zero-sum
Optimal Play	Always leads to draw or win for perfect AI
Complexity	O(b^d), grows exponentially
Strength	Ensures optimal decision
Weakness	Slow for large games without pruning
Improvement	Alpha-Beta pruning, depth-limiting, heuristic evaluation

---

✅ 8. Conclusion

The **Minimax algorithm** successfully determines the optimal move for Tic-Tac-Toe by simulating all possible game outcomes. It guarantees the best possible result for a perfect-play agent. However, the **computational complexity** increases exponentially with game size. Techniques like **Alpha-Beta pruning** and **heuristic evaluations** can significantly improve efficiency for larger games (like Chess or Connect-4).