**Date: 24th January 2025**

**Experiment 5**

**AIM:** Minimax (Flipped Class Assignment)

**Introduction:**

The **Minimax algorithm** is a decision-making algorithm used in game theory and artificial intelligence (AI), primarily in two-player games like chess or tic-tac-toe. Its main goal is to minimize the possible loss for a worst-case scenario while maximizing the player's best possible outcome. It's often used in competitive situations where both players are assumed to play optimally.

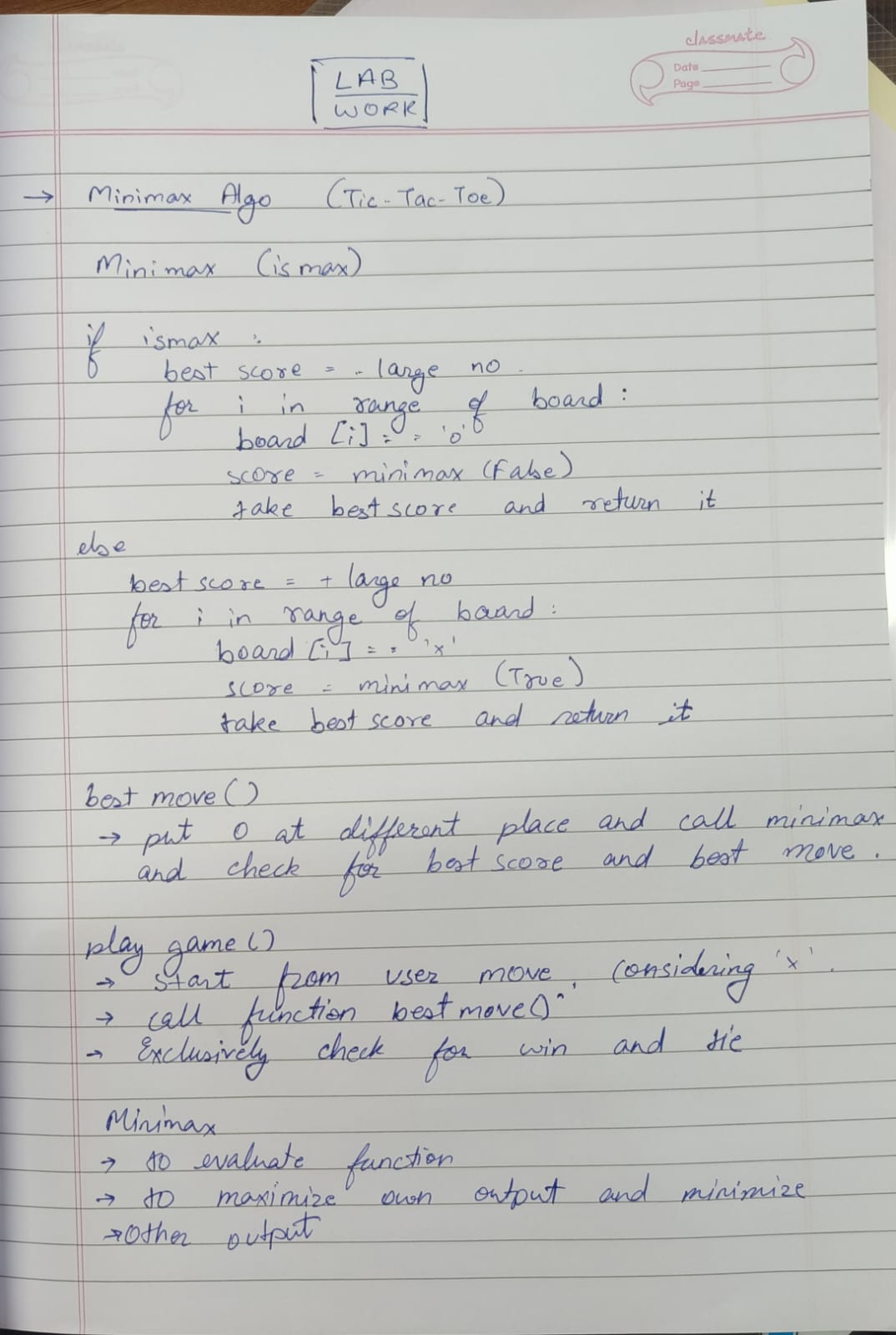
**How Minimax Works:**

1. **Tree Structure**: The game is represented as a tree where each node is a game state, and edges represent possible moves from one state to another.
2. **Alternating Turns**: The algorithm assumes that two players are taking turns. One player is trying to maximize their score (called the "maximizing player"), while the other is trying to minimize the maximizing player’s score (called the "minimizing player").
3. **Recursive Evaluation**:
   * The algorithm evaluates each possible move by recursively exploring all future moves in a depth-first manner.
   * At the terminal nodes (leaf nodes) of the tree, the algorithm assigns a value based on the outcome of the game (win, lose, or draw).
4. **Maximizing and Minimizing**:
   * If it's the maximizing player’s turn, the algorithm chooses the child node with the maximum value.
   * If it's the minimizing player’s turn, the algorithm chooses the child node with the minimum value.
5. **Backpropagation of Values**: The algorithm works backward from the terminal nodes to the root node (the current game state). It updates each node's value by considering the optimal move for each player at that point in the game.

**Steps of Minimax:**

1. Generate the game tree with all possible moves.
2. Evaluate terminal nodes using a utility function (e.g., +1 for win, -1 for loss, and 0 for draw).
3. Propagate the evaluations back to the root node by alternating between maximizing and minimizing at each level.
4. Choose the move that corresponds to the optimal value for the root node.

**Pseudo Code:**



**Code:**

|  |
| --- |
| ***TicTacToe*** |
| import math  EMPTY = ' '  **# Player representation**  PLAYER\_X = 'X'  # Program/PC/AI  PLAYER\_O = 'O'  # Human  **# To check for winner**  def winner(board):  **# For rows & columns**      for i in range(3):          if board[i][0] == board[i][1] == board[i][2] != EMPTY:              return board[i][0]          if board[0][i] == board[1][i] == board[2][i] != EMPTY:              return board[0][i]    **# For diagonals**      if board[0][0] == board[1][1] == board[2][2] != EMPTY:          return board[0][0]        if board[0][2] == board[1][1] == board[2][0] != EMPTY:          return board[0][2]        return None  **# To check if the board is full**  def isDraw(board):      for row in board:          for cell in row:              if cell == EMPTY:                  return False      return True  **# To check if the game is over**  def gameOver(board):      return winner(board) != None or isDraw(board)  **# To evaluate the game**  def evaluate(board):      if winner(board) == PLAYER\_X:          return 1      elif winner(board) == PLAYER\_O:          return -1      return 0  **# To get all available moves on the board**  def availableMoves(board):      moves = []      for i in range(3):          for j in range(3):              if board[i][j] == EMPTY:                  moves.append((i, j))      return moves  **# To make a move on the board**  def makeMove(board, move, player):      board[move[0]][move[1]] = player  **# To undo a move on the board**  def undoMove(board, move):      board[move[0]][move[1]] = EMPTY    **# Printing the board**  def printBoard(board):      count = 0      for row in board:          print(" | ".join(row))          count+=1          if count!=3:              print("----------")  **# Main Minimax algorithm ab aaya**  def minimax(board, depth, is\_maximizing\_player):      if gameOver(board):          return evaluate(board)        if is\_maximizing\_player:          bestScore = -math.inf          for move in availableMoves(board):              makeMove(board, move, PLAYER\_X)              eval = minimax(board, depth + 1, False)              undoMove(board, move)              bestScore = max(bestScore, eval)          return bestScore      else:          leastScore = math.inf          for move in availableMoves(board):              makeMove(board, move, PLAYER\_O)              eval = minimax(board, depth + 1, True)              undoMove(board, move)              leastScore = min(leastScore, eval)          return leastScore  **# To find the best move for program**  def findBestMove(board):      bestValue = -math.inf      bestMove = None        for move in availableMoves(board):          makeMove(board, move, PLAYER\_X)          moveValue = minimax(board, 0, False)          undoMove(board, move)            if moveValue > bestValue:              bestValue = moveValue              bestMove = move        return bestMove  **# Main function to play the game**  def playGame():      board = []      for i in range(3):          board.append([EMPTY] \* 3)      currentPlayer = PLAYER\_O        while not gameOver(board):          printBoard(board)            if currentPlayer == PLAYER\_X:              print("Program's turn (Player X):")              move = findBestMove(board)              makeMove(board, move, PLAYER\_X)              currentPlayer = PLAYER\_O          else:              print("Your turn (Player O):")              validMove = False              while not validMove:                  try:                      row = int(input("Enter row (0-2): "))                      col = int(input("Enter column (0-2): "))                      if board[row][col] == EMPTY:                          makeMove(board, (row, col), PLAYER\_O)                          validMove = True                      else:                          print("Cell already taken, try again.")                  except (ValueError, IndexError):                      print("Invalid move, try again.")              currentPlayer = PLAYER\_X        printBoard(board)        if winner(board) == PLAYER\_X:          print("You lost, better luck next time!")      elif winner(board) == PLAYER\_O:          print("You won!")      else:          print("It's a draw :)")  **# Entry Point into Algorithm**  playGame() |
| **Output:** |

**Conclusion:**

The Tic-Tac-Toe game implemented with the Minimax algorithm ensures that the AI plays optimally, making it impossible for the human player to win. The AI evaluates all possible moves to choose the best one, which either leads to a win or, in the worst case, a draw for the human player. As a result, the human player's best outcome is always a draw if they do not make mistakes. This project demonstrates how the Minimax algorithm guarantees optimal play and provides a solid foundation for more advanced AI applications in strategy games.