#MUHAMMAD HUZAIFA KHILJI

#f24-002

#LAB\_8

import math

class GameNode:

    """

    Represents a node in the game tree for Minimax.

    """

    def \_\_init\_\_(self, board, is\_maximizing, depth=0):

        self.board = board  # The current game state (e.g., Tic-Tac-Toe board)

        self.is\_maximizing = is\_maximizing  # True if the current player is maximizing

        self.children = []     # List of child GameNodes representing possible moves

        self.value = None      # The Minimax value of this node

        self.depth = depth

    def \_\_repr\_\_(self):

        return f"GameNode(maximizing={self.is\_maximizing}, value={self.value}, depth={self.depth})"

def minimax(node, depth, maximizing\_player):

    """

    Performs the Minimax algorithm to determine the best move.

    Args:

        node: The current GameNode.

        depth: The current depth in the game tree.

        maximizing\_player: Boolean indicating if it's the maximizing player's turn.

    Returns:

        The Minimax value of the node.

    """

    # Base case: If the game is over or the maximum depth is reached, return the evaluation of the board.

    if is\_game\_over(node.board) or depth == 0:

        return evaluate\_board(node.board)

    if maximizing\_player:

        best\_value = -math.inf  # Initialize to negative infinity for maximizing player

        for child in generate\_moves(node):

            value = minimax(child, depth - 1, False)  # Recursive call for the minimizing player

            best\_value = max(best\_value, value)  # Choose the maximum value

            node.value = best\_value

        return best\_value

    else:

        best\_value = math.inf  # Initialize to positive infinity for minimizing player

        for child in generate\_moves(node):

            value = minimax(child, depth - 1, True)  # Recursive call for the maximizing player

            best\_value = min(best\_value, value)  # Choose the minimum value

            node.value = best\_value

        return best\_value

def is\_game\_over(board):

    """

    Checks if the game is over (e.g., in Tic-Tac-Toe, if there's a winner or a tie).

    This function needs to be implemented based on the specific game.

    Args:

        board: The current game board state.

    Returns:

        True if the game is over, False otherwise.

    """

    # --- Tic-Tac-Toe Example (Replacewith your game's logic) ---

    # Check rows, columns, and diagonals for a win

    for row in board:

        if row[0] == row[1] == row[2] != ' ':

            return True

    for col in range(3):

        if board[0][col] == board[1][col] == board[2][col] != ' ':

            return True

    if board[0][0] == board[1][1] == board[2][2] != ' ':

        return True

    if board[0][2] == board[1][1] == board[2][0] != ' ':

        return True

    # Check for a tie

    for row in board:

        for cell in row:

            if cell == ' ':

                return False  # There's an empty cell, game is not over

    return True  # No empty cells, it's a tie

def evaluate\_board(board):

    """

    Evaluates the current game board state and returns a numerical score.

    This function needs to be implemented based on the specific game.

    A positive score is good for the maximizing player, a negative score is good

    for the minimizing player, and 0 is neutral.

    Args:

        board: The current game board state.

    Returns:

        A numerical score representing the evaluation of the board.

    """

    # --- Tic-Tac-Toe Example (Replace with your game's logic) ---

    #  'X' is the maximizing player, 'O' is the minimizing player

    for row in board:

        if row[0] == row[1] == row[2] != ' ':

            if row[0] == 'X':

                return 10

            else:

                return -10

    for col in range(3):

        if board[0][col] == board[1][col] == board[2][col] != ' ':

            if board[0][col] == 'X':

                return 10

            else:

                return -10

    if board[0][0] == board[1][1] == board[2][2] != ' ':

        if board[0][0] == 'X':

            return 10

        else:

            return -10

    if board[0][2] == board[1][1] == board[2][0] != ' ':

        if board[0][2] == 'X':

            return 10

        else:

            return -10

    return 0  # It's a tie

def generate\_moves(node):

    """

    Generates all possible moves (child nodes) from the current game state.

    This function needs to be implemented based on the specific game.

    Args:

        node: The current GameNode.

    Returns:

        A list of GameNode objects representing the possible next moves.

    """

    # --- Tic-Tac-Toe Example (Replace with your game's logic) ---

    moves = []

    board = node.board

    player = 'X' if node.is\_maximizing else 'O'

    for row in range(3):

        for col in range(3):

            if board[row][col] == ' ':

                new\_board = [row[:] for row in board]  # Create a copy of the board

                new\_board[row][col] = player          # Make the move

                moves.append(GameNode(new\_board, not node.is\_maximizing, node.depth + 1))  # Create a new GameNode

    return moves

# --- Example Usage (Tic-Tac-Toe) ---

# Initial game board (empty)

initial\_board = [

    [' ', ' ', ' '],

    [' ', ' ', ' '],

    [' ', ' ', ' ']

]

# Create the root node of the game tree

root\_node = GameNode(initial\_board, maximizing\_player=True, depth=0)

# Perform the Minimax algorithm to get the best move for the maximizing player ('X')

best\_move\_value = minimax(root\_node, 5, True) # depth 5 for example.

print("\nMinimax Algorithm:")

print(f"Best move value for the maximizing player (X): {best\_move\_value}")