

▼ THUẬT TOÁN MINIMAX

```
import copy
import math
import random
import numpy
X = "X"
O = "O"
EMPTY = None
user = None
ai = None
def initial_state():

    return [[EMPTY, EMPTY, EMPTY],
            [EMPTY, EMPTY, EMPTY],
            [EMPTY, EMPTY, EMPTY]]

def player (board) :
    x_count = 0
    o_count = 0
    for row in board:
        for cell in row:
            if cell == X:
                x_count += 1
            elif cell == O:
                o_count += 1

    if x_count == o_count:
        return X # X's turn
    else:
        return O # O's turn
```

initial_state(): Hàm này trả về trạng thái ban đầu của trò chơi cờ caro, đó là một bảng 3x3 với tất cả các ô đều trống (được điền EMPTY).

player(board): Hàm này xác định xem đến lượt của ai dựa trên số lượng quân cờ 'X' và 'O' trên bảng:

- Nếu số lượng 'X' và 'O' bằng nhau, thì đến lượt 'X' đi.
- Nếu số lượng 'X' nhiều hơn 'O', thì đến lượt 'O' đi.

```
def actions (board) :

    res = set ()
    board_len = len (board)
    for i in range (board_len) :
        for j in range (board_len) :
            if board[i] [j] == EMPTY:
                res.add ( (i, j) )
    return res
```

actions(board): Hàm này trả về một tập hợp các nước đi hợp lệ có thể thực hiện từ trạng thái bảng hiện tại. Nó duyệt qua tất cả các ô trên bảng và thêm tọa độ (i, j) của bất kỳ ô EMPTY nào vào tập hợp kết quả.

```
def result (board, action) :
    curr_player = player (board)
    result_board = copy. deepcopy (board)
    (i, j) = action
    result_board[i] [j] = curr_player
    return result_board
```

```
def get_horizontal_winner(board):
# check horizontally
winner_val = None
board_len = len(board)
for i in range(board_len):
    winner_val = board[i][0]
    # Check if all elements in the row are the same as the first element
    is_row_winner = True
```

```

for j in range(board_len):
    if board[i][j] != winner_val or winner_val == EMPTY:
        is_row_winner = False
        break
    if is_row_winner:
        return winner_val
return None

```

get_horizontal_winner(board): Hàm này kiểm tra xem có bất kỳ người chơi nào thắng cuộc theo hàng ngang hay không. Nó duyệt qua từng hàng trên bảng. Đối với mỗi hàng, nó kiểm tra xem tất cả các ô trong hàng đó có cùng một giá trị (X hoặc O) và không phải là EMPTY hay không. Nếu tìm thấy một hàng thắng, nó trả về biểu tượng của người chơi đó (X hoặc O). Nếu không, nó trả về None.

```

def get_vertical_winner (board) :
    # check vertically
    winner_val = None
    board_len = len (board)
    for i in range (board_len) :
        winner_val = board [0] [i]
        # Check if all elements in the column are the same as the first element
        is_col_winner = True
        for j in range (board_len) :
            if board[j][i] != winner_val or winner_val == EMPTY:
                is_col_winner = False
                break
        if is_col_winner:
            return winner_val
    return None

```

get_vertical_winner(board): Tương tự như get_horizontal_winner, hàm này kiểm tra xem có bất kỳ người chơi nào thắng cuộc theo hàng dọc hay không. Nó duyệt qua từng cột trên bảng và kiểm tra xem tất cả các ô trong cột đó có cùng một giá trị (X hoặc O) và không phải là EMPTY hay không. Nếu tìm thấy một cột thắng, nó trả về biểu tượng của người chơi đó. Nếu không, nó trả về None.

```

def get_diagonal_winner (board) :
    # check diagonally
    winner_val = None
    board_len = len (board)

    # Check main diagonal
    winner_val = board[0] [0]
    is_main_diag_winner = True
    for i in range (board_len) :
        if board[i] [i] != winner_val or winner_val == EMPTY:
            is_main_diag_winner = False
            break
    if is_main_diag_winner:
        return winner_val

    # Check anti-diagonal
    winner_val = board[0] [board_len - 1]
    is_anti_diag_winner = True
    for i in range (board_len) :
        j = board_len - 1 - i
        if board[i] [j] != winner_val or winner_val == EMPTY:
            is_anti_diag_winner = False
            break
    if is_anti_diag_winner:
        return winner_val

    return None

```

****get_diagonal_winner(board):**** Hàm này kiểm tra xem có người chơi nào thắng cuộc theo đường chéo hay không. Nó thực hiện hai kiểm tra:

- **Đường chéo chính:** Kiểm tra từ góc trên bên trái xuống góc dưới bên phải.
- **Đường chéo phụ:** Kiểm tra từ góc trên bên phải xuống góc dưới bên trái. Nếu một trong hai đường chéo có tất cả các ô cùng một giá trị (X hoặc O) và không phải là EMPTY, nó sẽ trả về biểu tượng của người chơi đó. Nếu không có người thắng theo đường chéo, nó trả về None.

```

def winner (board) :

    winner_val = get_horizontal_winner (board) or get_vertical_winner (board) or get_diagonal_winner (board)

```

```
return winner_val
```

```
def terminal(board):
    if winner(board) != None:
        return True
    for row in board:
        for cell in row:
            if cell == EMPTY:
                return False
    return True
```

Hàm này kiểm tra xem trạng thái bàn cờ hiện tại có phải là trạng thái cuối cùng của trò chơi hay không. Nó trả về True nếu có người thắng hoặc tất cả các ô trên bàn cờ đã được điền (hòa), và False nếu trò chơi vẫn có thể tiếp tục.

```
def utility(board):

    winner_val = winner(board)
    if winner_val == X:
        return 1
    elif winner_val == O:
        return -1
    return 0
```

Hàm này gán một giá trị số cho trạng thái cuối cùng (trạng thái kết thúc trò chơi) của bàn cờ. Nếu 'X' thắng, nó trả về 1. Nếu 'O' thắng, nó trả về -1. Nếu trò chơi hòa, nó trả về 0. Các hàm maxVal và minVal sử dụng giá trị này để đưa ra quyết định.

```
def maxVal(state):

    if terminal(state):
        return utility(state)
    v = -math.inf
    for action in actions(state):
        v = max(v, minVal(result(state, action)))
    return v
```

Hàm này được sử dụng khi đến lượt của người chơi tối đa hóa (AI đóng vai 'X'). Mục tiêu của nó là tìm nước đi dẫn đến giá trị lớn nhất có thể. Nó duyệt qua tất cả các nước đi hợp lệ từ trạng thái hiện tại, giả định rằng người chơi tiếp theo (minVal) sẽ chơi tối ưu để giảm thiểu giá trị.

```
def minVal(state):
    if terminal(state):
        return utility(state)
    v = math.inf
    for action in actions(state):
        v = min(v, maxVal(result(state, action)))
    return v
```

Hàm này được sử dụng khi đến lượt của người chơi tối thiểu hóa (AI đóng vai 'O', hoặc người chơi con người). Mục tiêu của nó là tìm nước đi dẫn đến giá trị nhỏ nhất có thể. Nó duyệt qua tất cả các nước đi hợp lệ từ trạng thái hiện tại, giả định rằng người chơi tiếp theo (maxVal) sẽ chơi tối ưu để tối đa hóa giá trị.

```
def minimax(board):

    current_player = player(board)
    if current_player == X:
        min_val = -math.inf
        move = None # Initialize move
        for action in actions(board):
            check = minVal(result(board, action)) # FIXED
            if check > min_val:
                min_val = check
                move = action
    else:
        max_val = math.inf
        move = None # Initialize move
        for action in actions(board):
            check = maxVal(result(board, action)) # FIXED
```

```

    if check < max_val:
        max_val = check
        move = action
    return move

```

hàm tìm nước đi tốt nhất (xem giá trị min và max value)

```

if __name__ == "__main__":
    board = initial_state()
    # ai_turn = False # Remove this variable
    print("Choose a player")
    user = input()
    if user == "X":
        ai = "O"
    else:
        ai = "X"

    while True:
        game_over = terminal(board)
        if game_over:
            winner_name = winner(board) # Renamed 'winner' to 'winner_name' to avoid conflict with function name
            if winner_name is None:
                print("Game Over: Tie.")
            else:
                print(f"Game Over: {winner_name} wins.")
            break

        current_player_symbol = player(board) # Get who is supposed to move (X or O)

        if user == current_player_symbol: # It's the human user's turn
            print("Enter the position to move (row,col)")
            i = int(input("Row:"))
            j = int(input("Col:"))
            if 0 <= i < 3 and 0 <= j < 3 and board[i][j] == EMPTY: # Added bounds check
                board = result(board, (i, j))
                print(numpy.array(board))
            else:
                print("Invalid move. Try again.")
        elif ai == current_player_symbol: # It's the AI's turn
            print("AI is making a move...") # Optional: give feedback
            move = minimax(board)
            board = result(board, move)
            print(numpy.array(board))

```

Choose a player

```

-----
KeyboardInterrupt                                Traceback (most recent call last)
/tmp/ipython-input-2128817363.py in <cell line: 0>()
      3 # ai_turn = False # Remove this variable
      4 print("Choose a player")
----> 5 user = input()
      6 if user == "X":
      7     ai = "O"

1 frames
/usr/local/lib/python3.12/dist-packages/ipykernel/kernelbase.py in _input_request(self, prompt, ident, parent, password)
    1217 except KeyboardInterrupt:
    1218     # re-raise KeyboardInterrupt, to truncate traceback
-> 1219     raise KeyboardInterrupt("Interrupted by user") from None
    1220 except Exception:
    1221     self.log.warning("Invalid Message:", exc_info=True)

KeyboardInterrupt: Interrupted by user

```

✓ SỬ DỤNG CẮT CẮT TỈA ALPHA BELTA

```

import os, math
def GetWinner(board):

    # horizontal
    if board[0] == board[1] and board[1] == board[2]:
        return board[0]
    elif board[3] == board[4] and board[4] == board[5]:
        return board[3]
    elif board[6] == board[7] and board[7] == board[8]:
        return board[6]

```

```
# vertical
elif board[0] == board[3] and board[3] == board[6]:
    return board[0]
elif board[1] == board[4] and board[4] == board[7]:
    return board[1]
elif board[2] == board[5] and board[5] == board[8]:
    return board[2] # Corrected indentation
# diagonal
elif board[0] == board[4] and board[4] == board[8]:
    return board[0] # Corrected indentation
elif board[2] == board[4] and board[4] == board[6]:
    return board[2] # Corrected indentation
return None # Explicitly return None if no winner
```

hàm **GetWinner(board)** trong phần cắt tỉa Alpha-Beta, chức năng của nó và cách nó xác định người chiến thắng trên một bảng 1D.

```
def PrintBoard(board):

    os.system('cls' if os.name=='nt' else 'clear')
    print(f'''
{board[0]}|{board[1]}|{board[2]}
{board[3]}|{board[4]}|{board[5]}
{board[6]}|{board[7]}|{board[8]}
''')
```

hàm **PrintBoard(board)** và cách nó hiển thị bảng Tic-Tac-Toe 1D để dễ đọc.

```
def GetAvailableCells(board):

    available = list()
    for cell in board:
        if cell != "X" and cell != "O":
            available.append(cell)
    return available
```

hàm **GetAvailableCells(board)** và cách nó xác định các ô trống có sẵn cho các nước đi tiếp theo.

```
def minimax(position, depth, alpha, beta, isMaximizing):

    winner = GetWinner(position)
    if winner != None:
        return 10 - depth if winner == "X" else -10 + depth
    if len(GetAvailableCells(position)) == 0:
        return 0 # Tie

    if isMaximizing:
        maxEval = -math.inf
        for cell in GetAvailableCells(position):
            position[cell - 1] = "X" # Try move for X (maximizing)
            Eval = minimax(position, depth + 1, alpha, beta, False) # Next turn is minimizing (O)
            maxEval = max(maxEval, Eval)
            alpha = max(alpha, Eval)
            position[cell - 1] = cell # Undo move
            if beta <= alpha:
                break # prune
        return maxEval # Moved outside the for loop
    else: # isMinimizing
        minEval = +math.inf
        for cell in GetAvailableCells(position):
            position[cell - 1] = "O" # Try move for O (minimizing)
            Eval = minimax(position, depth + 1, alpha, beta, True) # Next turn is maximizing (X)
            minEval = min(minEval, Eval)
            beta = min(beta, Eval)
            position[cell - 1] = cell # Undo move
            if beta <= alpha:
                break # prune
        return minEval # Moved outside the for loop
```

hàm **minimax(position, depth, alpha, beta, isMaximizing)** bao gồm các tham số, cách nó thực hiện đệ quy Minimax và cơ chế cắt tỉa Alpha-Beta (alpha, beta) để tối ưu hóa tìm kiếm.

```
def FindBestMove(currentPosition, AI):
    bestVal = -math.inf if AI == "X" else +math.inf
    bestMove = -1
    available_cells = GetAvailableCells(currentPosition)
```

```

# Nếu không còn nước đi, trả về -1
if not available_cells:
    return -1

for cell in available_cells:
    # 1. AI đi thử
    currentPosition[cell - 1] = AI

    # 2. Gọi Minimax cho lượt tiếp theo (LÀ LƯỢT CỦA ĐỐI THỦ)
    # SỬA Ở ĐÂY: Nếu AI là X, thì lượt sau là False (Minimizing).
    #           Nếu AI là O, thì lượt sau là True (Maximizing).
    is_ai_x = (AI == "X")
    moveVal = minimax(currentPosition, 0, -math.inf, +math.inf, not is_ai_x)

    # 3. Hoàn tác nước đi
    currentPosition[cell - 1] = cell

    # 4. Cập nhật điểm tốt nhất
    if (AI == "X" and moveVal > bestVal):
        bestMove = cell
        bestVal = moveVal
    elif (AI == "O" and moveVal < bestVal):
        bestMove = cell
        bestVal = moveVal

return bestMove

```

hàm `FindBestMove(currentPosition, AI)` và cách nó sử dụng hàm minimax (với Alpha-Beta) để xác định nước đi tối ưu cho AI, có tính đến vai trò của AI (tối đa hóa/tối thiểu hóa).

```

def main():
    player = input("Play as X or O? ").strip().upper()
    AI = "O" if player == "X" else "X"
    currentGame = [*range(1, 10)]
    # X always starts first.
    currentTurn = "X"
    counter = 0
    while True:
        if currentTurn == AI:
            # NOTE: if the AI starts first, it'll always choose index 0 so
            cell = FindBestMove(currentGame, AI)
            if cell == -1: # Handle case where FindBestMove might not find a move (though unlikely in Tic-Tac-Toe)
                print("AI could not find a move.")
                break
            currentGame[cell - 1] = AI
            currentTurn = player
        elif currentTurn == player:
            PrintBoard(currentGame)
            while True:
                try:
                    humanInput = int(input("Enter Number (1-9): ").strip())
                    if 1 <= humanInput <= 9 and humanInput in GetAvailableCells(currentGame): # Check if input is valid and cell is available
                        currentGame[humanInput - 1] = player
                        currentTurn = AI
                        break
                    else:
                        PrintBoard(currentGame)
                        print("Invalid input or cell not available. Try again.")
                except ValueError:
                    PrintBoard(currentGame)
                    print("Invalid input. Please enter a number.")

            # Check for winner or tie after each move
            if GetWinner(currentGame) != None:
                PrintBoard(currentGame)
                print(f"{GetWinner(currentGame)} WON!!!")
                break
            counter += 1
            if GetWinner(currentGame) == None and counter == 9: # All cells filled and no winner
                PrintBoard(currentGame)
                print("TIE")
                break

```

hàm `**main()` trong ngữ cảnh của phiên bản cắt tĩa Alpha-Beta, cách nó khởi tạo trò chơi, quản lý lượt chơi của người và AI, và xử lý kết thúc trò chơi.

```
if __name__ == "__main__":
    main()
```

Play as X or O? X

```
1|2|3
4|5|6
7|8|9
```

Enter Number (1-9): 6

```
1|2|0
4|5|X
7|8|9
```

Enter Number (1-9): 5

```
1|2|0
0|X|X
7|8|9
```

Enter Number (1-9): 8

```
1|0|0
0|X|X
7|X|9
```

Enter Number (1-9): 1

```
X|0|0
0|X|X
7|X|0
```

Enter Number (1-9): 7

```
X|0|0
0|X|X
X|X|0
```

TIE

Task

The user is encountering a logical error in their Tic-Tac-Toe AI where it fails to block a winning move. To identify the root cause, I will apply the Minimax algorithm debugging strategy. This involves analyzing the `minimax`, `maxValue`, `minValue`, `utility`, `winner`, `terminal`, `player`, `actions`, and `result` functions.

My plan is to:

1. **Consolidate Code:** Combine all the necessary Tic-Tac-Toe game logic (using the 2D board representation, as implied by previous interactions) into a single, cohesive block.
2. **Add Debugging Prints:** Insert `print` statements within the `minimax`, `maxValue`, and `minValue` functions. These prints will display the current board state, the player whose turn it is, the action being considered, the evaluated value for that action, and the `best_value` found so far. This will allow us to observe the AI's decision-making process step by step.
3. **Simulate Problematic Scenario:** I will run the game and manually input moves to create a scenario where the AI *should* block a user's winning move.
 - User (X) plays `(0,0)`.
 - AI (O) plays `(1,1)` (center).
 - User (X) plays `(0,1)`. At this point, the user has `X, X, _` in the top row.
 - It is now AI's turn. The AI *must* play `(0,2)` to block X from winning. If it chooses any other move, it indicates a bug.
4. **Analyze Debug Output:** Examine the output from the `print` statements during this scenario to pinpoint why the AI chose a suboptimal move instead of blocking. The root cause is often related to incorrect utility values, an incomplete search, or an error in how `max` or `min` values are selected.

By tracing the Minimax tree's evaluation for this critical juncture, I expect to identify the specific part of the algorithm that is miscalculating the value of the blocking move or incorrectly comparing it to other potential moves.

```
import copy
import math
import random
import numpy

# Constants for game pieces and empty cells
```

```

X = "X"
O = "O"
EMPTY = None

# Global variables for player assignments in the main game loop
user = None
ai = None

def initial_state():
    """
    Returns starting state of the board.
    A 3x3 board with all cells empty.
    """
    return [[EMPTY, EMPTY, EMPTY],
            [EMPTY, EMPTY, EMPTY],
            [EMPTY, EMPTY, EMPTY]]

def player(board):
    """
    Returns player who has the next turn on a board.
    Counts X's and O's to determine whose turn it is.
    """
    x_count = 0
    o_count = 0
    for row in board:
        for cell in row:
            if cell == X:
                x_count += 1
            elif cell == O:
                o_count += 1

    if x_count == o_count:
        return X # X always starts, and if counts are equal, it's X's turn
    else:
        return O # If X count is greater, it's O's turn

def actions(board):
    """
    Returns set of all possible actions (i, j) available on the board.
    An action is a tuple (row, column) representing an empty cell.
    """
    res = set()
    board_len = len(board)
    for i in range(board_len):
        for j in range(board_len):
            if board[i][j] == EMPTY:
                res.add((i, j))
    return res

def result(board, action):
    """
    Returns the board that results from making move (i, j) on the board.
    Raises ValueError if the action is invalid.
    """
    # Determine current player based on board state
    curr_player = player(board)

    # Create a deep copy of the board to avoid modifying the original
    result_board = copy.deepcopy(board)

    (i, j) = action

    # Check if the move is valid
    if not (0 <= i < len(board) and 0 <= j < len(board[0]) and result_board[i][j] == EMPTY):
        raise ValueError("Invalid action: cell is already taken or out of bounds.")

    # Apply the move

```

```

    result_board[i][j] = curr_player
    return result_board

# Helper functions to check for a winner in rows, columns, and diagonals
def get_horizontal_winner(board):
    """Checks for a winner in any row."""
    board_len = len(board)
    for i in range(board_len):
        winner_val = board[i][0]
        is_row_winner = True
        if winner_val == EMPTY: # If the first cell is empty, it cannot be a winner yet
            is_row_winner = False
        else:
            for j in range(1, board_len): # Start from the second element
                if board[i][j] != winner_val:
                    is_row_winner = False
                    break
            if is_row_winner:
                return winner_val
    return None

def get_vertical_winner(board):
    """Checks for a winner in any column."""
    board_len = len(board)
    for i in range(board_len):
        winner_val = board[0][i]
        is_col_winner = True
        if winner_val == EMPTY: # If the first cell is empty, it cannot be a winner yet
            is_col_winner = False
        else:
            for j in range(1, board_len): # Start from the second element
                if board[j][i] != winner_val:
                    is_col_winner = False
                    break
            if is_col_winner:
                return winner_val
    return None

def get_diagonal_winner(board):
    """Checks for a winner in either diagonal."""
    board_len = len(board)

    # Check main diagonal (top-left to bottom-right)
    winner_val_main = board[0][0]
    is_main_diag_winner = True
    if winner_val_main == EMPTY:
        is_main_diag_winner = False
    else:
        for i in range(1, board_len):
            if board[i][i] != winner_val_main:
                is_main_diag_winner = False
                break
    if is_main_diag_winner:
        return winner_val_main

    # Check anti-diagonal (top-right to bottom-left)
    winner_val_anti = board[0][board_len - 1]
    is_anti_diag_winner = True
    if winner_val_anti == EMPTY:
        is_anti_diag_winner = False
    else:
        for i in range(1, board_len):
            j = board_len - 1 - i
            if board[i][j] != winner_val_anti:
                is_anti_diag_winner = False
                break
    if is_anti_diag_winner:
        return winner_val_anti

    return None

```

```

def winner(board):
    """
    Returns the winner of the game, if there is one.
    X if X wins, O if O wins, None otherwise.
    """
    # Check all winning conditions
    winner_val = get_horizontal_winner(board) or \
        get_vertical_winner(board) or \
        get_diagonal_winner(board)
    return winner_val

def terminal(board):
    """
    Returns True if game is over, False otherwise.
    Game is over if there is a winner or if all cells are filled (tie).
    """
    if winner(board) is not None:
        return True
    # Check if there are any empty cells left
    for row in board:
        for cell in row:
            if cell == EMPTY:
                return False # Game is not over if there's an empty cell
    return True # No winner and no empty cells, it's a tie

def utility(board):
    """
    Returns 1 if X has won the game, -1 if O has won, 0 otherwise (tie).
    """
    winner_val = winner(board)
    if winner_val == X:
        return 1
    elif winner_val == O:
        return -1
    return 0 # Tie or game not over

def maxValue(state):
    """
    Calculates the maximum value for the Maximizing player (X).
    This function is part of the Minimax algorithm.
    """
    print(f" [maxValue] Evaluating board for X:\n{numpy.array(state)}") # DEBUG
    if terminal(state):
        util_val = utility(state)
        print(f" [maxValue] Terminal state, utility: {util_val}") # DEBUG
        return util_val

    v = -math.inf
    for action in actions(state):
        print(f" [maxValue] X considers action: {action}") # DEBUG
        current_v = minValue(result(state, action))
        v = max(v, current_v)
        print(f" [maxValue] After action {action}, current max value for X: {v}") # DEBUG
    return v

def minValue(state):
    """
    Calculates the minimum value for the Minimizing player (O).
    This function is part of the Minimax algorithm.
    """
    print(f" [minValue] Evaluating board for O:\n{numpy.array(state)}") # DEBUG
    if terminal(state):
        util_val = utility(state)
        print(f" [minValue] Terminal state, utility: {util_val}") # DEBUG
        return util_val

```

```

v = math.inf
for action in actions(state):
    print(f"    [minValue] 0 considers action: {action}") # DEBUG
    current_v = maxValue(result(state, action))
    v = min(v, current_v)
    print(f"    [minValue] After action {action}, current min value for 0: {v}") # DEBUG
return v

def minimax(board):
    """
    Returns the optimal action for the current player on the board.
    This is the top-level function that initiates the Minimax search.
    """
    print(f"\n[minimax] Starting Minimax search for player {player(board)} on board:\n{numpy.array(board)}") # DEBUG
    current_player = player(board)
    best_move = None

    if current_player == X: # Maximizing player
        best_value = -math.inf
        for action in actions(board):
            print(f"    [minimax] {current_player} considering action: {action}") # DEBUG
            # Simulate the action and get the value from the minimizing player's perspective
            value = minValue(result(board, action))
            print(f"    [minimax] Value for action {action}: {value}") # DEBUG
            if value > best_value:
                best_value = value
                best_move = action
        print(f"[minimax] {current_player} chose move {best_move} with value {best_value}") # DEBUG
    else: # Minimizing player (current_player == O)
        best_value = math.inf
        for action in actions(board):
            print(f"    [minimax] {current_player} considering action: {action}") # DEBUG
            # Simulate the action and get the value from the maximizing player's perspective
            value = maxValue(result(board, action))
            print(f"    [minimax] Value for action {action}: {value}") # DEBUG
            if value < best_value:
                best_value = value
                best_move = action
        print(f"[minimax] {current_player} chose move {best_move} with value {best_value}") # DEBUG
    return best_move

# Main game loop
if __name__ == "__main__":
    board = initial_state()
    print("Choose a player (X/O):")
    user_input_player = input().strip().upper()

    if user_input_player == "X":
        user = "X"
        ai = "O"
    elif user_input_player == "O":
        user = "O"
        ai = "X"
    else:
        print("Invalid choice. Defaulting to User as X, AI as O.")
        user = "X"
        ai = "O"

    print(f"You are {user}, AI is {ai}.")

    while True:
        game_over = terminal(board)
        if game_over:
            winner_name = winner(board)
            if winner_name is None:
                print("\nGame Over: It's a Tie!")
            else:
                print(f"\nGame Over: {winner_name} wins!")

```

```

        break

current_player_symbol = player(board)

if user == current_player_symbol: # Human user's turn
    print(f"\nYour turn ({user}).")
    print("Current board:")
    print(numpy.array(board))
    while True:
        try:
            row = int(input("Enter row (0, 1, or 2): "))
            col = int(input("Enter column (0, 1, or 2): "))
            action = (row, col)

            if 0 <= row < 3 and 0 <= col < 3 and board[row][col] == EMPTY:
                board = result(board, action)
                break
            else:
                print("Invalid move. Cell is already taken or out of bounds. Try again.")
        except ValueError:
            print("Invalid input. Please enter numbers for row and column.")
    elif ai == current_player_symbol: # AI's turn
        print(f"\nAI's turn ({ai}).")
        print("Current board:")
        print(numpy.array(board))
        print("AI is making a move...")
        move = minimax(board)
        if move is not None:
            board = result(board, move)
        else:
            print("AI could not find a valid move. This should not happen in Tic-Tac-Toe unless the board is full.")
            break

```

Analyze Minimax and Helper Functions

Subtask:

Examine the `minimax`, `maxValue`, `minValue`, `utility`, `winner`, `terminal`, `player`, `actions`, and `result` functions for any potential logical flaws or incorrect implementations of the Minimax algorithm or board evaluation.

Analyze Minimax and Helper Functions

Subtask:

Examine the `minimax`, `maxValue`, `minValue`, `utility`, `winner`, `terminal`, `player`, `actions`, and `result` functions for any potential logical flaws or incorrect implementations of the Minimax algorithm or board evaluation.

Instructions

- Review the `winner` function and its helper functions (`get_horizontal_winner`, `get_vertical_winner`, `get_diagonal_winner`) to confirm they accurately identify all winning conditions for both 'X' and 'O', paying close attention to how `EMPTY` cells are handled in win checks.
- Examine the `terminal` function to ensure it correctly determines when the game ends, either by a win/loss or a tie.
- Verify the `utility` function assigns the correct numerical values: 1 for an 'X' win, -1 for an 'O' win, and 0 for a tie.
- Check the `player`, `actions`, and `result` functions to ensure they correctly identify the current player, valid moves, and generate the subsequent board state after a move.
- Focus on the core Minimax functions: `minimax`, `maxValue`, and `minValue`. Ensure that:
 - `maxValue` correctly aims to maximize the score and appropriately calls `minValue` for the opponent's turn.
 - `minValue` correctly aims to minimize the score and appropriately calls `maxValue` for the opponent's turn.
 - The base cases for both `maxValue` and `minValue` correctly return the `utility` value when a `terminal` state is reached.
 - The top-level `minimax` function correctly selects the `best_value` and `best_move` based on whether the current player is maximizing ('X') or minimizing ('O').
 - Confirm that the debugging `print` statements, which were added by the previous agent, are present in `minimax`, `maxValue`, and `minValue` as they will be crucial for the next debugging steps.

Analyze Minimax and Helper Functions

Subtask:

Examine the `minimax`, `maxValue`, `minValue`, `utility`, `winner`, `terminal`, `player`, `actions`, and `result` functions for any potential logical flaws or incorrect implementations of the Minimax algorithm or board evaluation.

Instructions

1. Review the `winner` function and its helper functions (`get_horizontal_winner`, `get_vertical_winner`, `get_diagonal_winner`) to confirm they accurately identify all winning conditions for both 'X' and 'O', paying close attention to how `EMPTY` cells are handled in win checks.
2. Examine the `terminal` function to ensure it correctly determines when the game ends, either by a win/loss or a tie.
3. Verify the `utility` function assigns the correct numerical values: 1 for an 'X' win, -1 for an 'O' win, and 0 for a tie.
4. Check the `player`, `actions`, and `result` functions to ensure they correctly identify the current player, valid moves, and generate the subsequent board state after a move.
5. Focus on the core Minimax functions: `minimax`, `maxValue`, and `minValue`. Ensure that:
 - `maxValue` correctly aims to maximize the score and appropriately calls `minValue` for the opponent's turn.
 - `minValue` correctly aims to minimize the score and appropriately calls `maxValue` for the opponent's turn.
 - The base cases for both `maxValue` and `minValue` correctly return the `utility` value when a `terminal` state is reached.
 - The top-level `minimax` function correctly selects the `best_value` and `best_move` based on whether the current player is maximizing ('X') or minimizing ('O').
 - Confirm that the debugging `print` statements, which were added by the previous agent, are present in `minimax`, `maxValue`, and `minValue` as they will be crucial for the next debugging steps.

Analyze Minimax and Helper Functions

Subtask:

Examine the `minimax`, `maxValue`, `minValue`, `utility`, `winner`, `terminal`, `player`, `actions`, and `result` functions for any potential logical flaws or incorrect implementations of the Minimax algorithm or board evaluation.

Instructions

1. Review the `winner` function and its helper functions (`get_horizontal_winner`, `get_vertical_winner`, `get_diagonal_winner`) to confirm they accurately identify all winning conditions for both 'X' and 'O', paying close attention to how `EMPTY` cells are handled in win checks.
2. Examine the `terminal` function to ensure it correctly determines when the game ends, either by a win/loss or a tie.
3. Verify the `utility` function assigns the correct numerical values: 1 for an 'X' win, -1 for an 'O' win, and 0 for a tie.
4. Check the `player`, `actions`, and `result` functions to ensure they correctly identify the current player, valid moves, and generate the subsequent board state after a move.
5. Focus on the core Minimax functions: `minimax`, `maxValue`, and `minValue`. Ensure that:
 - `maxValue` correctly aims to maximize the score and appropriately calls `minValue` for the opponent's turn.
 - `minValue` correctly aims to minimize the score and appropriately calls `maxValue` for the opponent's turn.
 - The base cases for both `maxValue` and `minValue` correctly return the `utility` value when a `terminal` state is reached.
 - The top-level `minimax` function correctly selects the `best_value` and `best_move` based on whether the current player is maximizing ('X') or minimizing ('O').
 - Confirm that the debugging `print` statements, which were added by the previous agent, are present in `minimax`, `maxValue`, and `minValue` as they will be crucial for the next debugging steps.

Analyze Minimax and Helper Functions

Subtask:

Examine the `minimax`, `maxValue`, `minValue`, `utility`, `winner`, `terminal`, `player`, `actions`, and `result` functions for any potential logical flaws or incorrect implementations of the Minimax algorithm or board evaluation.

Instructions

1. Review the `winner` function and its helper functions (`get_horizontal_winner`, `get_vertical_winner`, `get_diagonal_winner`) to confirm they accurately identify all winning conditions for both 'X' and 'O', paying close attention to how `EMPTY` cells are handled in win checks.
2. Examine the `terminal` function to ensure it correctly determines when the game ends, either by a win/loss or a tie.
3. Verify the `utility` function assigns the correct numerical values: 1 for an 'X' win, -1 for an 'O' win, and 0 for a tie.
4. Check the `player`, `actions`, and `result` functions to ensure they correctly identify the current player, valid moves, and generate the subsequent board state after a move.
5. Focus on the core Minimax functions: `minimax`, `maxValue`, and `minValue`. Ensure that:
 - `maxValue` correctly aims to maximize the score and appropriately calls `minValue` for the opponent's turn.
 - `minValue` correctly aims to minimize the score and appropriately calls `maxValue` for the opponent's turn.
 - The base cases for both `maxValue` and `minValue` correctly return the `utility` value when a `terminal` state is reached.
 - The top-level `minimax` function correctly selects the `best_value` and `best_move` based on whether the current player is maximizing ('X') or minimizing ('O').
 - Confirm that the debugging `print` statements, which were added by the previous agent, are present in `minimax`, `maxValue`, and `minValue` as they will be crucial for the next debugging steps.

Analyze Minimax and Helper Functions

Subtask:

Examine the `minimax`, `maxValue`, `minValue`, `utility`, `winner`, `terminal`, `player`, `actions`, and `result` functions for any potential logical flaws or incorrect implementations of the Minimax algorithm or board evaluation.

Instructions

1. Review the `winner` function and its helper functions (`get_horizontal_winner`, `get_vertical_winner`, `get_diagonal_winner`) to confirm they accurately identify all winning conditions for both 'X' and 'O', paying close attention to how `EMPTY` cells are handled in win checks.
2. Examine the `terminal` function to ensure it correctly determines when the game ends, either by a win/loss or a tie.
3. Verify the `utility` function assigns the correct numerical values: 1 for an 'X' win, -1 for an 'O' win, and 0 for a tie.
4. Check the `player`, `actions`, and `result` functions to ensure they correctly identify the current player, valid moves, and generate the subsequent board state after a move.
5. Focus on the core Minimax functions: `minimax`, `maxValue`, and `minValue`. Ensure that:
 - `maxValue` correctly aims to maximize the score and appropriately calls `minValue` for the opponent's turn.
 - `minValue` correctly aims to minimize the score and appropriately calls `maxValue` for the opponent's turn.
 - The base cases for both `maxValue` and `minValue` correctly return the `utility` value when a `terminal` state is reached.
 - The top-level `minimax` function correctly selects the `best_value` and `best_move` based on whether the current player is maximizing ('X') or minimizing ('O').
 - Confirm that the debugging `print` statements, which were added by the previous agent, are present in `minimax`, `maxValue`, and `minValue` as they will be crucial for the next debugging steps.

Analyze Minimax and Helper Functions

Subtask:

Examine the `minimax`, `maxValue`, `minValue`, `utility`, `winner`, `terminal`, `player`, `actions`, and `result` functions for any potential logical flaws or incorrect implementations of the Minimax algorithm or board evaluation.

Instructions

1. Review the `winner` function and its helper functions (`get_horizontal_winner`, `get_vertical_winner`, `get_diagonal_winner`) to confirm they accurately identify all winning conditions for both 'X' and 'O', paying close attention to how `EMPTY` cells are handled in win checks.
2. Examine the `terminal` function to ensure it correctly determines when the game ends, either by a win/loss or a tie.
3. Verify the `utility` function assigns the correct numerical values: 1 for an 'X' win, -1 for an 'O' win, and 0 for a tie.
4. Check the `player`, `actions`, and `result` functions to ensure they correctly identify the current player, valid moves, and generate the subsequent board state after a move.
5. Focus on the core Minimax functions: `minimax`, `maxValue`, and `minValue`. Ensure that:
 - `maxValue` correctly aims to maximize the score and appropriately calls `minValue` for the opponent's turn.

- `minValue` correctly aims to minimize the score and appropriately calls `maxValue` for the opponent's turn.
- The base cases for both `maxValue` and `minValue` correctly return the `utility` value when a `terminal` state is reached.
- The top-level `minimax` function correctly selects the `best_value` and `best_move` based on whether the current player is maximizing ('X') or minimizing ('O').
- Confirm that the debugging `print` statements, which were added by the previous agent, are present in `minimax`, `maxValue`, and `minValue` as they will be crucial for the next debugging steps.

Analyze Minimax and Helper Functions

Subtask:

Examine the `minimax`, `maxValue`, `minValue`, `utility`, `winner`, `terminal`, `player`, `actions`, and `result` functions for any potential logical flaws or incorrect implementations of the Minimax algorithm or board evaluation.

Instructions

1. Review the `winner` function and its helper functions (`get_horizontal_winner`, `get_vertical_winner`, `get_diagonal_winner`) to confirm they accurately identify all winning conditions for both 'X' and 'O', paying close attention to how `EMPTY` cells are handled in win checks.
2. Examine the `terminal` function to ensure it correctly determines when the game ends, either by a win/loss or a tie.
3. Verify the `utility` function assigns the correct numerical values: 1 for an 'X' win, -1 for an 'O' win, and 0 for a tie.
4. Check the `player`, `actions`, and `result` functions to ensure they correctly identify the current player, valid moves, and generate the subsequent board state after a move.
5. Focus on the core Minimax functions: `minimax`, `maxValue`, and `minValue`. Ensure that:
 - `maxValue` correctly aims to maximize the score and appropriately calls `minValue` for the opponent's turn.
 - `minValue` correctly aims to minimize the score and appropriately calls `maxValue` for the opponent's turn.
 - The base cases for both `maxValue` and `minValue` correctly return the `utility` value when a `terminal` state is reached.
 - The top-level `minimax` function correctly selects the `best_value` and `best_move` based on whether the current player is maximizing ('X') or minimizing ('O').
 - Confirm that the debugging `print` statements, which were added by the previous agent, are present in `minimax`, `maxValue`, and `minValue` as they will be crucial for the next debugging steps.

Analyze Minimax and Helper Functions

Subtask:

Examine the `minimax`, `maxValue`, `minValue`, `utility`, `winner`, `terminal`, `player`, `actions`, and `result` functions for any potential logical flaws or incorrect implementations of the Minimax algorithm or board evaluation.

Instructions

1. Review the `winner` function and its helper functions (`get_horizontal_winner`, `get_vertical_winner`, `get_diagonal_winner`) to confirm they accurately identify all winning conditions for both 'X' and 'O', paying close attention to how `EMPTY` cells are handled in win checks.
2. Examine the `terminal` function to ensure it correctly determines when the game ends, either by a win/loss or a tie.
3. Verify the `utility` function assigns the correct numerical values: 1 for an 'X' win, -1 for an 'O' win, and 0 for a tie.
4. Check the `player`, `actions`, and `result` functions to ensure they correctly identify the current player, valid moves, and generate the subsequent board state after a move.
5. Focus on the core Minimax functions: `minimax`, `maxValue`, and `minValue`. Ensure that:
 - `maxValue` correctly aims to maximize the score and appropriately calls `minValue` for the opponent's turn.
 - `minValue` correctly aims to minimize the score and appropriately calls `maxValue` for the opponent's turn.
 - The base cases for both `maxValue` and `minValue` correctly return the `utility` value when a `terminal` state is reached.
 - The top-level `minimax` function correctly selects the `best_value` and `best_move` based on whether the current player is maximizing ('X') or minimizing ('O').
 - Confirm that the debugging `print` statements, which were added by the previous agent, are present in `minimax`, `maxValue`, and `minValue` as they will be crucial for the next debugging steps.

Analyze Minimax and Helper Functions

Subtask:

Examine the `minimax`, `maxValue`, `minValue`, `utility`, `winner`, `terminal`, `player`, `actions`, and `result` functions for any potential logical flaws or incorrect implementations of the Minimax algorithm or board evaluation.

Instructions

1. Review the `winner` function and its helper functions (`get_horizontal_winner`, `get_vertical_winner`, `get_diagonal_winner`) to confirm they accurately identify all winning conditions for both 'X' and 'O', paying close attention to how `EMPTY` cells are handled in win checks.
2. Examine the `terminal` function to ensure it correctly determines when the game ends, either by a win/loss or a tie.
3. Verify the `utility` function assigns the correct numerical values: 1 for an 'X' win, -1 for an 'O' win, and 0 for a tie.
4. Check the `player`, `actions`, and `result` functions to ensure they correctly identify the current player, valid moves, and generate the subsequent board state after a move.
5. Focus on the core Minimax functions: `minimax`, `maxValue`, and `minValue`. Ensure that:
 - `maxValue` correctly aims to maximize the score and appropriately calls `minValue` for the opponent's turn.
 - `minValue` correctly aims to minimize the score and appropriately calls `maxValue` for the opponent's turn.
 - The base cases for both `maxValue` and `minValue` correctly return the `utility` value when a `terminal` state is reached.
 - The top-level `minimax` function correctly selects the `best_value` and `best_move` based on whether the current player is maximizing ('X') or minimizing ('O').
 - Confirm that the debugging `print` statements, which were added by the previous agent, are present in `minimax`, `maxValue`, and `minValue` as they will be crucial for the next debugging steps.

Analyze Minimax and Helper Functions

Subtask:

Examine the `minimax`, `maxValue`, `minValue`, `utility`, `winner`, `terminal`, `player`, `actions`, and `result` functions for any potential logical flaws or incorrect implementations of the Minimax algorithm or board evaluation.

Instructions

1. Review the `winner` function and its helper functions (`get_horizontal_winner`, `get_vertical_winner`, `get_diagonal_winner`) to confirm they accurately identify all winning conditions for both 'X' and 'O', paying close attention to how `EMPTY` cells are handled in win checks.
2. Examine the `terminal` function to ensure it correctly determines when the game ends, either by a win/loss or a tie.
3. Verify the `utility` function assigns the correct numerical values: 1 for an 'X' win, -1 for an 'O' win, and 0 for a tie.
4. Check the `player`, `actions`, and `result` functions to ensure they correctly identify the current player, valid moves, and generate the subsequent board state after a move.
5. Focus on the core Minimax functions: `minimax`, `maxValue`, and `minValue`. Ensure that:
 - `maxValue` correctly aims to maximize the score and appropriately calls `minValue` for the opponent's turn.
 - `minValue` correctly aims to minimize the score and appropriately calls `maxValue` for the opponent's turn.
 - The base cases for both `maxValue` and `minValue` correctly return the `utility` value when a `terminal` state is reached.
 - The top-level `minimax` function correctly selects the `best_value` and `best_move` based on whether the current player is maximizing ('X') or minimizing ('O').
 - Confirm that the debugging `print` statements, which were added by the previous agent, are present in `minimax`, `maxValue`, and `minValue` as they will be crucial for the next debugging steps.

Analyze Minimax and Helper Functions

Subtask:

Examine the `minimax`, `maxValue`, `minValue`, `utility`, `winner`, `terminal`, `player`, `actions`, and `result` functions for any potential logical flaws or incorrect implementations of the Minimax algorithm or board evaluation.

Instructions

1. Review the `winner` function and its helper functions (`get_horizontal_winner`, `get_vertical_winner`, `get_diagonal_winner`) to confirm they accurately identify all winning conditions for both 'X' and 'O', paying close attention to how `EMPTY` cells are handled in win checks.
2. Examine the `terminal` function to ensure it correctly determines when the game ends, either by a win/loss or a tie.
3. Verify the `utility` function assigns the correct numerical values: 1 for an 'X' win, -1 for an 'O' win, and 0 for a tie.
4. Check the `player`, `actions`, and `result` functions to ensure they correctly identify the current player, valid moves, and generate the subsequent board state after a move.
5. Focus on the core Minimax functions: `minimax`, `maxValue`, and `minValue`. Ensure that:
 - `maxValue` correctly aims to maximize the score and appropriately calls `minValue` for the opponent's turn.
 - `minValue` correctly aims to minimize the score and appropriately calls `maxValue` for the opponent's turn.
 - The base cases for both `maxValue` and `minValue` correctly return the `utility` value when a `terminal` state is reached.
 - The top-level `minimax` function correctly selects the `best_value` and `best_move` based on whether the current player is maximizing ('X') or minimizing ('O').
 - Confirm that the debugging `print` statements, which were added by the previous agent, are present in `minimax`, `maxValue`, and `minValue` as they will be crucial for the next debugging steps.

Analyze Minimax and Helper Functions

Subtask:

Examine the `minimax`, `maxValue`, `minValue`, `utility`, `winner`, `terminal`, `player`, `actions`, and `result` functions for any potential logical flaws or incorrect implementations of the Minimax algorithm or board evaluation.

Instructions

1. Review the `winner` function and its helper functions (`get_horizontal_winner`, `get_vertical_winner`, `get_diagonal_winner`) to confirm they accurately identify all winning conditions for both 'X' and 'O', paying close attention to how `EMPTY` cells are handled in win checks.
2. Examine the `terminal` function to ensure it correctly determines when the game ends, either by a win/loss or a tie.
3. Verify the `utility` function assigns the correct numerical values: 1 for an 'X' win, -1 for an 'O' win, and 0 for a tie.
4. Check the `player`, `actions`, and `result` functions to ensure they correctly identify the current player, valid moves, and generate the subsequent board state after a move.
5. Focus on the core Minimax functions: `minimax`, `maxValue`, and `minValue`. Ensure that:
 - `maxValue` correctly aims to maximize the score and appropriately calls `minValue` for the opponent's turn.
 - `minValue` correctly aims to minimize the score and appropriately calls `maxValue` for the opponent's turn.
 - The base cases for both `maxValue` and `minValue` correctly return the `utility` value when a `terminal` state is reached.
 - The top-level `minimax` function correctly selects the `best_value` and `best_move` based on whether the current player is maximizing ('X') or minimizing ('O').
 - Confirm that the debugging `print` statements, which were added by the previous agent, are present in `minimax`, `maxValue`, and `minValue` as they will be crucial for the next debugging steps.

Analyze Minimax and Helper Functions

Subtask:

Examine the `minimax`, `maxValue`, `minValue`, `utility`, `winner`, `terminal`, `player`, `actions`, and `result` functions for any potential logical flaws or incorrect implementations of the Minimax algorithm or board evaluation.

Instructions

1. Review the `winner` function and its helper functions (`get_horizontal_winner`, `get_vertical_winner`, `get_diagonal_winner`) to confirm they accurately identify all winning conditions for both 'X' and 'O', paying close attention to how `EMPTY` cells are handled in win checks.
2. Examine the `terminal` function to ensure it correctly determines when the game ends, either by a win/loss or a tie.
3. Verify the `utility` function assigns the correct numerical values: 1 for an 'X' win, -1 for an 'O' win, and 0 for a tie.
4. Check the `player`, `actions`, and `result` functions to ensure they correctly identify the current player, valid moves, and generate the subsequent board state after a move.
5. Focus on the core Minimax functions: `minimax`, `maxValue`, and `minValue`. Ensure that:
 - `maxValue` correctly aims to maximize the score and appropriately calls `minValue` for the opponent's turn.

- `minValue` correctly aims to minimize the score and appropriately calls `maxValue` for the opponent's turn.
- The base cases for both `maxValue` and `minValue` correctly return the `utility` value when a `terminal` state is reached.
- The top-level `minimax` function correctly selects the `best_value` and `best_move` based on whether the current player is maximizing ('X') or minimizing ('O').
- Confirm that the debugging `print` statements, which were added by the previous agent, are present in `minimax`, `maxValue`, and `minValue` as they will be crucial for the next debugging steps.

Analyze Minimax and Helper Functions

Subtask:

Examine the `minimax`, `maxValue`, `minValue`, `utility`, `winner`, `terminal`, `player`, `actions`, and `result` functions for any potential logical flaws or incorrect implementations of the Minimax algorithm or board evaluation.

Instructions

1. Review the `winner` function and its helper functions (`get_horizontal_winner`, `get_vertical_winner`, `get_diagonal_winner`) to confirm they accurately identify all winning conditions for both 'X' and 'O', paying close attention to how `EMPTY` cells are handled in win checks.
2. Examine the `terminal` function to ensure it correctly determines when the game ends, either by a win/loss or a tie.
3. Verify the `utility` function assigns the correct numerical values: 1 for an 'X' win, -1 for an 'O' win, and 0 for a tie.
4. Check the `player`, `actions`, and `result` functions to ensure they correctly identify the current player, valid moves, and generate the subsequent board state after a move.
5. Focus on the core Minimax functions: `minimax`, `maxValue`, and `minValue`. Ensure that:
 - `maxValue` correctly aims to maximize the score and appropriately calls `minValue` for the opponent's turn.
 - `minValue` correctly aims to minimize the score and appropriately calls `maxValue` for the opponent's turn.
 - The base cases for both `maxValue` and `minValue` correctly return the `utility` value when a `terminal` state is reached.
 - The top-level `minimax` function correctly selects the `best_value` and `best_move` based on whether the current player is maximizing ('X') or minimizing ('O').
 - Confirm that the debugging `print` statements, which were added by the previous agent, are present in `minimax`, `maxValue`, and `minValue` as they will be crucial for the next debugging steps.

Analyze Minimax and Helper Functions

Subtask:

Examine the `minimax`, `maxValue`, `minValue`, `utility`, `winner`, `terminal`, `player`, `actions`, and `result` functions for any potential logical flaws or incorrect implementations of the Minimax algorithm or board evaluation.

Instructions

1. Review the `winner` function and its helper functions (`get_horizontal_winner`, `get_vertical_winner`, `get_diagonal_winner`) to confirm they accurately identify all winning conditions for both 'X' and 'O', paying close attention to how `EMPTY` cells are handled in win checks.
2. Examine the `terminal` function to ensure it correctly determines when the game ends, either by a win/loss or a tie.
3. Verify the `utility` function assigns the correct numerical values: 1 for an 'X' win, -1 for an 'O' win, and 0 for a tie.
4. Check the `player`, `actions`, and `result` functions to ensure they correctly identify the current player, valid moves, and generate the subsequent board state after a move.
5. Focus on the core Minimax functions: `minimax`, `maxValue`, and `minValue`. Ensure that:
 - `maxValue` correctly aims to maximize the score and appropriately calls `minValue` for the opponent's turn.
 - `minValue` correctly aims to minimize the score and appropriately calls `maxValue` for the opponent's turn.
 - The base cases for both `maxValue` and `minValue` correctly return the `utility` value when a `terminal` state is reached.
 - The top-level `minimax` function correctly selects the `best_value` and `best_move` based on whether the current player is maximizing ('X') or minimizing ('O').
 - Confirm that the debugging `print` statements, which were added by the previous agent, are present in `minimax`, `maxValue`, and `minValue` as they will be crucial for the next debugging steps.

Analyze Minimax and Helper Functions

Subtask:

Examine the `minimax`, `maxValue`, `minValue`, `utility`, `winner`, `terminal`, `player`, `actions`, and `result` functions for any potential logical flaws or incorrect implementations of the Minimax algorithm or board evaluation.

Instructions

1. Review the `winner` function and its helper functions (`get_horizontal_winner`, `get_vertical_winner`, `get_diagonal_winner`) to confirm they accurately identify all winning conditions for both 'X' and 'O', paying close attention to how `EMPTY` cells are handled in win checks.
2. Examine the `terminal` function to ensure it correctly determines when the game ends, either by a win/loss or a tie.
3. Verify the `utility` function assigns the correct numerical values: 1 for an 'X' win, -1 for an 'O' win, and 0 for a tie.
4. Check the `player`, `actions`, and `result` functions to ensure they correctly identify the current player, valid moves, and generate the subsequent board state after a move.
5. Focus on the core Minimax functions: `minimax`, `maxValue`, and `minValue`. Ensure that:
 - `maxValue` correctly aims to maximize the score and appropriately calls `minValue` for the opponent's turn.
 - `minValue` correctly aims to minimize the score and appropriately calls `maxValue` for the opponent's turn.
 - The base cases for both `maxValue` and `minValue` correctly return the `utility` value when a `terminal` state is reached.
 - The top-level `minimax` function correctly selects the `best_value` and `best_move` based on whether the current player is maximizing ('X') or minimizing ('O').
 - Confirm that the debugging `print` statements, which were added by the previous agent, are present in `minimax`, `maxValue`, and `minValue` as they will be crucial for the next debugging steps.

Simulate Problematic Scenario

Subtask:

Run the game and manually input moves to create a scenario where the AI should block a user's winning move. Observe the debug output to understand the AI's decision-making process.

Reasoning: The subtask requires running the existing Tic-Tac-Toe game code to simulate a specific scenario. The `if __name__ == "__main__": main()` block is responsible for initiating the game loop and handling user input and AI moves.