ASSIGNMENT 4

SESSION 1

def play_player(board, player):

```
def play_player(board, player):
    """Allow the player to make a move on the board."""
    def check_p(board, column):
        length = len(board)
        counter = -1
        for i in range(length):
            counter += 1
            if board[i][column] != " ":
                return counter

    column = user_input(board)
    if check_p(board, column) == None:
        board[-1][column] = player
    else:
        y = int(check_p(board, column) - 1)
        board[y][column] = player
    return board
```

Purpose
Updates the board to reflect the player's move in the specified column. The move is placed in the lowest available row of that column.

Inputs
We pass the board of the game, and which player we are, either ‘X’ or ‘O’

---

def check_p(board, column):

Purpose
We have another function inside which determines the row index in the specified column where the player's move can be placed. It works in the following way :

- Iterates through the rows of the board from bottom to top.
- When it encounters a non-empty space (!= “ “) it stops and returns its index. This effectively identifies the first available spot above this occupied space.
- If the column is completely empty, it returns ‘None’ because the counter is not incremented for a sequence of empty spaces.

If check_p returns None, it means the column is entirely empty. Hence, the player's marker is placed in the last row of that column.
Else, if check_p returns a valid index, the player’s move is placed in the first available row above the highest occupied space.

Finally, the updated board is returned, reflecting the player's move.

---

def play_random_computer(board, player):

Purpose
To simulate the computer's turn in the game by having the computer randomly select a column from those that are not full and place its marker in the appropriate position.

Inputs
board: of the game
player: computer’s marker ‘X’ or ‘O’.

```
def play_random_computer(board, player):
    """Computer plays randomly, choosing an empty column to place its marker."""
    # TODO: Implement function for computer's turn where it plays randomly
    # Ensure the selected column is not full before placing the marker
    # Update the board with the computer's move and then print the board
    cols = len(board[0])
    # Find all columns that are not full
    available_columns = []
    for col in range(cols):
        if board[0][col] == " ":  # Check if the top cell of the column is empty
            available_columns.append(col)  # Add column to available columns if not full

    # Choose a random column from the available ones
    if available_columns:
        chosen_column = random.choice(available_columns)

        # Find the lowest empty row in the chosen column
        for row in range(len(board) - 1, -1, -1):  # Start from the bottom row
            if board[row][chosen_column] == " ":
                board[row][chosen_column] = player  # Place the marker
                break  # Exit after placing the marker
    else:
        print("No moves left for the computer!")
```

It iterates through the columns to identify those which are not full:

- A column is considered ‘not full’ if the highest cell (board[0][col]) is empty.
- If so, we add the available columns to the list available_columns .

If there are available columns, a random column is selected using random.choice().

Once a column is selected :

- The function iterates through the rows in that column, starting from the bottom (len(board) - 1) and moving upwards (`-1` is the stop condition, and `-1` is the step).
- It will place the computer's marker (`player`) in the lowest empty row of the chosen column.
- break is there to ensure that only one marker is placed.

If there are no available columns, the computer announces that it cannot make a move.

---

def play_heuristic_computer(board, player):

Purpose

The function chooses the best possible move for the computer by checking for moves that allow the computer to win immediately;

Inputs

board: of the game
player: computer’s marker ‘X’ or ‘O’.

```python
def play_heuristic_computer(board, player):
    """Computer plays based on a heuristic: prioritize center columns and block potential wins."""

    # Function to check if placing in a column results in a win for the player
    def check_win_movement(board, player):
        for row_idx, row in enumerate(board):
            for col_idx in range(len(board[0])):
                if row[col_idx] == ' ':
                    # Copy board to test placement
                    test_board = [r[:] for r in board]
                    test_board[row_idx][col_idx] = player

                    # Check for win after placement
                    if check_win(test_board, player):
                        return col_idx
        return None

    # Attempt to play winning or blocking move
    col = check_win_movement(board, player)
    if col is not None:
        return make_move(board, col, player)

    # Attempt to block opponent
    opponent = 'X' if player == 'O' else 'O'
    col = check_win_movement(board, opponent)
    if col is not None:
        return make_move(board, col, player)

    # Play center columns if no immediate winning or blocking move is found
    col = play_center_columns(board)
    if col is not None:
        return make_move(board, col, player)

    draw_board(board)  # Display the board after computer's move
```

We have another function inside which checks if placing a marker in any column would result in a win for the given player. It works the following way :

- It checks over the cells in the board (row_idx, col_idx).
- If the cell is empty, it simulates placing the marker on that cell using a test board.
- It then uses check_win to see if this simulated move results in a win.
- If a winning column is found, it returns the column index (col_idx); otherwise, it returns None.

We have **3 heuristics** :

1. Attempt a winning move : If check_win_movement returns a column index, the computer plays in that column using make_move which places the marker in the appropriate row of the specified column.
2. Block opponent's win : We use check_win_movement with the opponents marker, if a blocking move is found the computer places its marker in that column.
3. Prioritize center columns : The function play_center_columns is explained in the following section.

We then display the updated board.

---

def play_center_columns(board, player):

Purpose

Select an available column near the center of the board.

Inputs

board: board of the game
col: optional argument (we do not use it in this implementation).

```python
def play_center_columns(board, col=None):
    """Choose an available column near the center, prioritizing middle columns."""
    cols = len(board[0])
    center_col = cols // 2

    # Create a list of column indices ordered from the center outwards
    column_order = [center_col] + [center_col + i for i in range(1, (cols + 1) // 2)]
    column_order += [center_col - i for i in range(1, (cols + 1) // 2)]

    # Return the first available column from the prioritized list
    for col in column_order:
        if board[0][col] == " ":
            return col

    return None  # No columns available
```

We calculate the total number of columns, and identify the index of the central column (cols // 2).

We generate a list of column indices starting with the central column and expanding outwards. We start with center_col, then add the indices of columns to the right of the center, and add indices of columns to the left of the center.
Presented as :
Center → Columns to the right → Columns to the left (in alternating order)

We find the first available column by iterating over the ordered list of column indices, checking the topmost cell to see if it is empty and returning the first available column from the prioritized list.

If there are no columns available we return None.

---

def make_move(board, player):

Purpose
Ensures the marker is placed in the lowest available row of the specified column.

Inputs
board: of the game
col: index of the column
player: computer's marker 'X' or 'O'.

The loop iterates over the rows of the board from bottom to top as explained in play_random_computer, in a way that it simulates gravity.

- We check if the cell is empty, if so we place the marker in that position.
- We return the column index where the move was made.

```
def make_move(board, col, player):
    """Place the player's marker in the chosen column."""
    for row in range(len(board) - 1, -1, -1):
        if board[row][col] == " ":
            board[row][col] = player
            return col
```

---

def is_moves_left(board):

Purpose
Check whether there are any available moves left on the board.

Inputs
board: of the game

```
def is_moves_left(board, ):
    """Check if there are any moves left on the board."""

    for row in board:
        if ' ' in row:
            return True

    return False
```

It iterates through the rows in the board, if an empty cell is found it returns True, else it returns False.

---

def check_win(board, player):

Purpose
Determines whether a specified player has won the game by forming a sequence of four consecutive markers on the board.

Inputs
board: of the game
player: computer's marker 'X' or 'O'.

```
def check_win(board, player):
    """Check if the specified player has won the game."""
    # TODO: Implement win check logic
    # Check all vertical, horizontal and diagonal lines
    # Return True if the player has won, otherwise False
    rows = len(board)
    cols = len(board[0])
```

We first calculate the number of columns and rows in the board.

Horizontal Win

```
# Check horizontal wins
for row in range(rows):
    for col in range(cols - 3):  # Only go up to the 4th-last column
        # Check if there are four consecutive 'player' markers horizontally
        if (board[row][col] == player and
                board[row][col + 1] == player and
                board[row][col + 2] == player and
                board[row][col + 3] == player):
            return True
```

Iterates through each row and checks for consecutive markers starting from each column up to the 4th-last column, to prevent index overflow.

If there are 4 consecutive cells in the row containing the player's marker, it returns True.

Vertical Win

```
# Check vertical wins
for col in range(cols):
    for row in range(rows - 3):  # Only go up to the 4th-last row
        # Check if there are four consecutive 'player' markers vertically
        if (board[row][col] == player and
                board[row + 1][col] == player and
                board[row + 2][col] == player and
                board[row + 3][col] == player):
            return True
```

Iterates through each column and checks for consecutive markers starting from each row up to the 4th-last row, to prevent index overflow.

If there are 4 consecutive cells in the column containing the player's marker, it returns True.

Diagonal Win (Bottom-Left to Top-Right)

```
# Check diagonal wins (bottom-left to top-right)
for row in range(rows - 3):
    for col in range(cols - 3):
        # Check if there are four consecutive 'player' markers diagonally
        if (board[row][col] == player and
                board[row + 1][col + 1] == player and
                board[row + 2][col + 2] == player and
                board[row + 3][col + 3] == player):
            return True
```

It starts from each cell where a diagonal of four can fit. Iterates rows up to the 4th-last row and columns up to the 4th-last column.

If there are 4 consecutive cells in the diagonal containing the player's marker, it returns True.

Diagonal Win (Top-Left to Bottom-Right)

```
# Check diagonal wins (top-left to bottom-right)
for row in range(3, rows):
    for col in range(cols - 3):
        # Check if there are four consecutive 'player' markers diagonally
        if (board[row][col] == player and
                board[row - 1][col + 1] == player and
                board[row - 2][col + 2] == player and
                board[row - 3][col + 3] == player):
            return True
```

It starts from each cell where a diagonal of four can fit. Iterates the rows starting from row 3 onwards, and up to the 4th-last column.

If there are 4 consecutive cells in the diagonal containing the player's marker, it returns True.

If there is no win found, it returns False.

SESSION 2

def heuristic_score(board, player):

Purpose

Determine how favorable the board is for the player based on positions that increase their chances of winning or block the opponent.

Inputs

board: of the game
player: computer's marker 'X' or 'O'.

We prioritize the center column as it connects more potential winning lines.

The function counts how many times the player's marker appears in the center column and multiplies it by 3 to give additional weight.

```python
def heuristic_score(board, player):
    """Calculate the heuristic score for the board based on the player."""
    score = 0
    opponent = 'X' if player == 'O' else 'O'  # Identify the opponent's marker.

    # Score the center column (column index 3)
    # Center column is often strategically advantageous as it connects more potential winning lines.
    center_array = [row[3] for row in board]
    center_count = center_array.count(player)
    score += center_count * 3  # Give extra weight to center positions for the player.

    # Evaluate horizontal windows of 4 cells
    for row in board:
        for c in range(7 - 3):  # Iterate through all possible horizontal groups of 4.
            window = row[c:c + 4]
            score += evaluate_window(window, player)  # Evaluate the group for the player's advantage.

    # Evaluate vertical windows of 4 cells
    for c in range(7):  # Loop through each column.
        col_array = [board[r][c] for r in range(6)]  # Extract the entire column as a list.
        for r in range(6 - 3):  # Iterate through all possible vertical groups of 4.
            window = col_array[r:r + 4]
            score += evaluate_window(window, player)  # Evaluate the group for the player's advantage.

    # Evaluate positive diagonal windows of 4 cells
    # A positive diagonal goes from top-left to bottom-right.
    for r in range(6 - 3):  # Rows where a diagonal of 4 can start.
        for c in range(7 - 3):  # Columns where a diagonal of 4 can start.
            window = [board[r + i][c + i] for i in range(4)]  # Extract the diagonal group of 4.
            score += evaluate_window(window, player)  # Evaluate the diagonal for the player's advantage

    # Evaluate negative diagonal windows of 4 cells
    # A negative diagonal goes from bottom-left to top-right.
```

To evaluate horizontal, vertical, positive diagonal or negative diagonal windows we use aid from the following function :

def evaluate_window(window, player):

Purpose

Assign a score to a "window" that reflects how favorable it is for a given player.

Inputs

window: list of 4 cells that represent a part of the board
player: player's marker 'X' or 'O'.

```python
def evaluate_window(window, player):
    """Evaluate a window of four cells and assign a score."""
    score = 0
    opponent = 'X' if player == 'O' else 'O'  # Identify the opponent's marker.

    # Check for winning conditions for the player
    if window.count(player) == 4:  # If all 4 cells are occupied by the player, it's a winning group.
        score += 100
    elif window.count(player) == 3 and window.count(' ') == 1:  # 3 player markers and 1 empty space.
        score += 5
    elif window.count(player) == 2 and window.count(' ') == 2:  # 2 player markers and 2 empty spaces.
        score += 2

    # Check for potential blocks for the opponent
    if window.count(opponent) == 3 and window.count(' ') == 1:  # 3 opponent markers and 1 empty space.
        score -= 4  # Subtract points to prioritize blocking the opponent.

    return score
```

If 4 cells are occupied, it's a winning sequence and we award a high score (+100). If 3 cells are occupied, the player is one step away from winning so we make it high-priority (+5). If there are 2 occupied cells, it's less urgent (+2) but there is still a chance of winning. On the other hand, we also check potential blocks for our opponent.

def minmax_algorithm(*):

```python
def minmax_algorithm(board, depth, maximizing_player, player, heuristic=False):
    """
    Basic Minimax algorithm to calculate the best move for the current player.
    """
    valid_locations = get_valid_locations(board)  # Get all columns where a move is possible.

    # Base case: If depth is 0, no valid moves remain, or a win condition is met, evaluate the board.
    if depth == 0 or not valid_locations or check_win(board, player) or check_win(board, 'X' if player == 'O' else 'O'):
        return None, evaluation_score(board, player, heuristic=heuristic)

    opponent = 'X' if player == 'O' else 'O'  # Determine the opponent's marker.

    if maximizing_player:
        value = -math.inf  # Initialize the best value as negative infinity.
        best_column = random.choice(valid_locations)  # Pick a random valid column to start.
        for col in valid_locations:
            temp_board = [row[:] for row in board]  # Create a copy of the board to simulate the move.
            make_move(temp_board, col, player)  # Simulate the player's move in the column.
            # Recursively call Minimax for the minimizing player (opponent).
            new_score = minmax_algorithm(temp_board, depth - 1, maximizing_player: False, player, heuristic)[1]
            if new_score > value:  # If the new score is better, update the best value and column.
                value = new_score
                best_column = col
        return best_column, value  # Return the best column and its score.
    else:
        value = math.inf  # Initialize the best value as positive infinity.
        best_column = random.choice(valid_locations)  # Pick a random valid column to start.
        for col in valid_locations:
            temp_board = [row[:] for row in board]  # Create a copy of the board to simulate the move.
            make_move(temp_board, col, opponent)  # Simulate the opponent's move in the column.
            # Recursively call Minimax for the maximizing player.
            new_score = minmax_algorithm(temp_board, depth - 1, maximizing_player: True, player, heuristic)[1]
```

Purpose

The minmax_algorithm function determines the best move in a Connect Four game using the **Minimax** algorithm. It evaluates potential future game states recursively up to a specified depth and alternates between the maximizing and minimizing players to find the optimal move.

Parameters

board: Current game state (2D list) representing the Connect Four grid.

depth: Maximum number of moves ahead to simulate in the search tree.

maximizing_player: Boolean value indicating whether the current player is trying to maximize the score (True) or minimize the score (False).

player: The current player's marker ('X' or 'O') to be used for evaluation.

heuristic: Boolean flag indicating whether to use heuristic scoring for evaluation. If True, a heuristic evaluation is used; otherwise, a simple evaluation is used.

Function Logic

Base Case The recursion halts If depth == 0, meaning the search has reached the maximum depth, if there are no valid columns to drop a piece (i.e., the list of valid columns is empty), if a win condition is detected for either player.

When any of these conditions occur, the function returns the evaluation score of the current board by calling the evaluation_score() function.

Maximizing Player

The algorithm simulates all possible valid moves for the maximizing player by iterating through the valid columns where a move can be made.

For each valid move: A temporary copy of the board is made, and the current move is simulated using the make_move() function and the function recursively calls minmax_algorithm() for the minimizing player (the opponent) to evaluate the resulting game state. The best score for the maximizing player is tracked, and the column with the highest

score is selected.

Minimizing Player

The algorithm simulates all possible valid moves for the minimizing player by iterating through the valid columns where the opponent can make a move.

For each valid move, a temporary copy of the board is made, and the current move is simulated using the make_move() function. The function recursively calls minmax_algorithm() for the maximizing player to evaluate the resulting game state. The worst score for the maximizing player is tracked, and the column that minimizes this score is selected.

Output

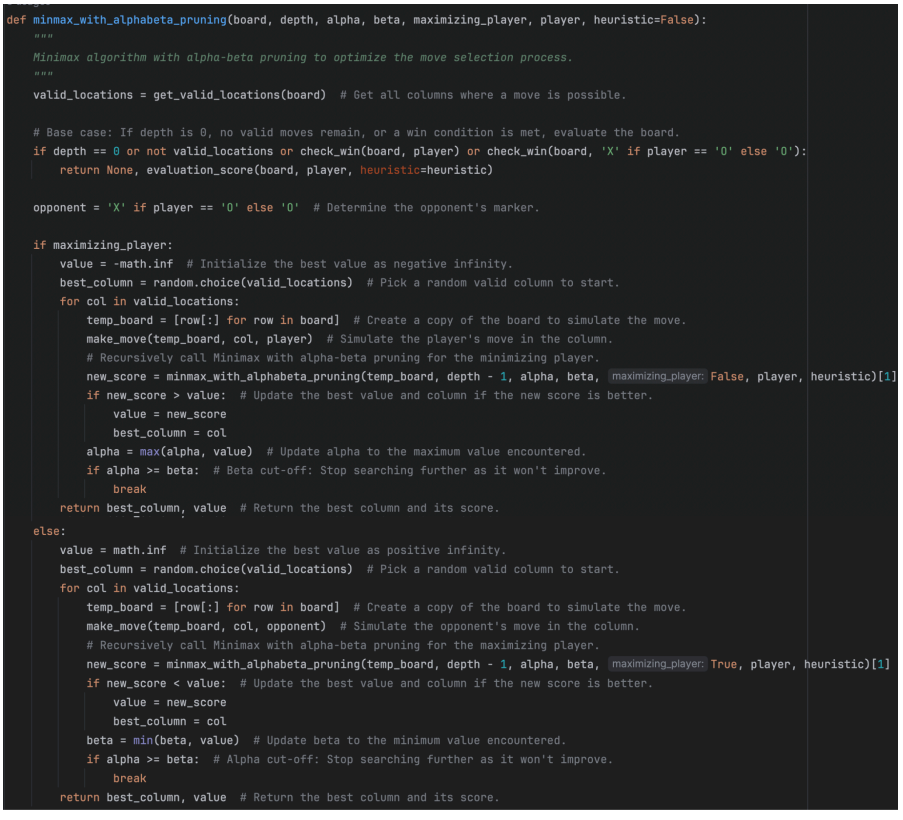
The function returns the **best column** to play, along with its associated **evaluation score**.

---

def minmax_with_alphabeta_pruning(*):

Purpose

The minmax_with_alphabeta_pruning function optimizes the Minimax algorithm by using **alpha-beta pruning**, reducing the number of unnecessary node evaluations during the decision-making process.



```
def minmax_with_alphabeta_pruning(board, depth, alpha, beta, maximizing_player, player, heuristic=False):
    """
    Minimax algorithm with alpha-beta pruning to optimize the move selection process.
    """
    valid_locations = get_valid_locations(board)  # Get all columns where a move is possible.

    # Base case: If depth is 0, no valid moves remain, or a win condition is met, evaluate the board.
    if depth == 0 or not valid_locations or check_win(board, player) or check_win(board, 'X' if player == 'O' else 'O'):
        return None, evaluation_score(board, player, heuristic=heuristic)

    opponent = 'X' if player == 'O' else 'O'  # Determine the opponent's marker.

    if maximizing_player:
        value = -math.inf  # Initialize the best value as negative infinity.
        best_column = random.choice(valid_locations)  # Pick a random valid column to start.
        for col in valid_locations:
            temp_board = [row[:] for row in board]  # Create a copy of the board to simulate the move.
            make_move(temp_board, col, player)  # Simulate the player's move in the column.
            # Recursively call Minimax with alpha-beta pruning for the minimizing player.
            new_score = minmax_with_alphabeta_pruning(temp_board, depth - 1, alpha, beta, maximizing_player: False, player, heuristic)[1]
            if new_score > value:  # Update the best value and column if the new score is better.
                value = new_score
                best_column = col
            alpha = max(alpha, value)  # Update alpha to the maximum value encountered.
            if alpha >= beta:  # Beta cut-off: Stop searching further as it won't improve.
                break
        return best_column, value  # Return the best column and its score.
    else:
        value = math.inf  # Initialize the best value as positive infinity.
        best_column = random.choice(valid_locations)  # Pick a random valid column to start.
        for col in valid_locations:
            temp_board = [row[:] for row in board]  # Create a copy of the board to simulate the move.
            make_move(temp_board, col, opponent)  # Simulate the opponent's move in the column.
            # Recursively call Minimax with alpha-beta pruning for the maximizing player.
            new_score = minmax_with_alphabeta_pruning(temp_board, depth - 1, alpha, beta, maximizing_player: True, player, heuristic)[1]
            if new_score < value:  # Update the best value and column if the new score is better.
                value = new_score
                best_column = col
            beta = min(beta, value)  # Update beta to the minimum value encountered.
            if alpha >= beta:  # Alpha cut-off: Stop searching further as it won't improve.
                break
        return best_column, value  # Return the best column and its score.
```

Inputs

board: Current game state (2D list).

depth: Maximum depth for the search tree.

alpha: Best value achievable for the maximizing player (initially -inf).

beta: Best value achievable for the minimizing player (initially +inf).

maximizing_player: Whether the current player is maximizing the score.

player: Current player's marker ('X' or 'O').

heuristic: Use heuristic scoring for evaluation if True.

Function Logic

Base Case: Stops recursion if depth is 0, no valid moves exist, or a win condition is detected.

Returns the board's evaluation score.

Maximizing Player: Simulates all valid moves and recursively evaluates the opponent's responses and updates alpha to track the best achievable score and performs a **beta cut-off** (stops exploring further moves when alpha >= beta).

Minimizing Player: Simulates all valid moves, evaluates the maximizing player's responses, and updates beta to track the worst-case scenario for the opponent. Performs an alpha **cut-off** (stops exploring further moves when alpha >= beta).

Output

Returns the best column to play and its associated score.

---

def play_computer_minmax(*):

Purpose

The play_computer_minmax function determines the best move for the computer player in a Connect Four game using either the basic Minimax algorithm or the Alpha-Beta pruning optimization, depending on the use_pruning flag. It simulates possible moves and selects the best column for the computer to play, considering the current state of the board and maximizing the player's score.

```python
def play_computer_minmax(board, player, use_pruning=False, heuristic=False):
    """
    Use the Minimax or Alpha-Beta pruning algorithm to determine the computer's move.
    """
    depth = 6  # Set the search depth to limit computation time.
    if use_pruning:
        # Determine the best column using Alpha-Beta pruning.
        col, minimax_score = minmax_with_alphabeta_pruning(board, depth, -math.inf, math.inf, True, player, heuristic)
    else:
        # Determine the best column using basic Minimax.
        col, minimax_score = minmax_algorithm(board, depth, True, player, heuristic)
    if col is not None:
        make_move(board, col, player)  # Make the move in the chosen column.
    else:
        print("No valid moves available for the computer!")  # Handle cases where no valid moves exist.
```

Inputs

board: The current game state represented as a 2D list (game board).

player: The computer's marker ('X' or 'O') used to make the decision.

use_pruning: A boolean flag indicating whether to use Alpha-Beta pruning for optimization (True) or basic Minimax (False).

heuristic: A boolean flag that specifies whether to use a heuristic scoring function for the evaluation (True), or not (False).

Function Logic

Base Case: The function checks if the current game state is valid for making a move. If no valid moves are available, it will print an error message stating "No valid moves available for the computer!"

Alpha-Beta Pruning: If use_pruning is True, the function uses the minmax_with_alphabeta_pruning() function to determine the best column. This implementation incorporates Alpha-Beta pruning to reduce unnecessary node evaluations in the decision tree, speeding up the search for the best move.

It calls the minmax_with_alphabeta_pruning() function with the current board state, a search depth of 6, initial values for alpha and beta, and the current player.

Basic Minimax: If use_pruning is False, the function falls back to the basic Minimax algorithm, calling minmax_algorithm() with the same parameters.

Output

The function returns the best column to play (if a valid move exists). The column is chosen based on the evaluation of the possible moves at the current depth of the game tree.

If no valid move exists (i.e., the board is full), it will print "No valid moves available for the computer!"

Heuristics evaluation

Throughout the weeks we have been working with different heuristics:

Random heuristic

One simple heuristic favoring center columns

Board evaluation heuristic

The heuristic we have been developing through Session 2 is the **board evaluation heuristic**. This heuristic has demonstrated **superior performance**, since it analyzes the board and checks for **center-placed pieces, winning moves and consecutive-placed pieces** through **windowing analysis of the board** . It then assigns the corresponding score to each configuration (essentially yielding higher scores for boards in which the player is closer to connect 4).

This strategy, as previously mentioned, has translated into a **smarter computer-player**. If the computer plays first, there is no chance for a human to win .

Additional comments

Regarding the depth parameter, this is defined within play_computer_minmax(*) (the caller function for min_max & make_move functions).
Following Davide's advice, we tested **up to which depth was 'worth it'** computationally speaking. In other words, since increasing the depth parameter enlarges the **computational**

effort of the algorithm, we assessed up to which value there were **significant improvements in performance**, so that we could choose a value that led to enhanced "intelligence" of our algorithm **without overwhelming the CPU**. The final value we decided following this procedure was 6.
Nevertheless, keep in mind that **the best way to take this decision** would be to **mathematically prove** the most **optimal value for depth**.

PEER EVALUATION

- Josep Cubedo
    - Team assessment participation:
        - Adriana Martinez: /10
        - Oscar Thiele: /10
        - Julian Harder: /10
- Oscar Thiele
    - Team assessment participation:
        - Adriana Martinez: /10
        - Josep Cubedo: /10
        - Julian Harder: /10
- Julian Harder
    - Team assessment participation:
        - Adriana Martinez: /10
        - Oscar Thiele: /10
        - Josep Cubedo: /10
- Adriana Martinez
    - Team assessment participation:
        - Oscar Thiele: 10/10
        - Josep Cubedo: 10/10
        - Julian Harder: 10/10