CONNECT FOUR AI REPORT

Connect Four is a classic two-player game where players drop colored discs into a grid to connect four of their own in a row, either vertically, horizontally, or diagonally. The game features an AI opponent utilizing the minimax with alpha-beta pruning algorithm for strategic decision-making.

1. Evaluation Function:

The evaluation function is a crucial component of the AI strategy in Connect Four. It assigns a score to each possible game state, indicating the desirability of that state for the AI player. The function needs to capture the game dynamics and prioritize moves that lead to a winning position while avoiding those that allow the opponent to win.

1.1 Line Evaluation:

The evaluation function assesses the significance of a line of consecutive pieces, considering the counts of AI and opponent pieces in that line. Different counts lead to different scores. For example, having four consecutive AI pieces results in a very high score, indicating a potential victory, while having three consecutive AI pieces is also favorable but to a lesser extent.

```
def evaluate_line(line_values, max_player):


    counts = {
        4: {max_player: 1000, 3 - max_player: -1000},
        3: {max_player: 100, 3 - max_player: -100},
        2: {max_player: 10, 3 - max_player: -10}
    }

    ai_count = line_values.tolist().count(max_player)
    opponent_count = line_values.tolist().count(3 - max_player)
```

1.2 Board Position Evaluation:

The overall board evaluation considers the sum of scores for all possible lines in rows, columns, and diagonals. It assigns positive infinity if the AI player has already won, negative infinity if the opponent has won, and a score based on the individual line evaluations otherwise.

```python
def evaluate_position(board, max_player):
    winner = board.check_victory()

    if winner == max_player: return inf
    elif winner != 0: return -inf

    score = 0
    grid = board.grid

    for line in range(6):
        score += np.sum(evaluate_line(grid[:, line][i:i+4], max_player) for i in range(4))

    for column in range(7):
        score += np.sum(evaluate_line(grid[column, :][i:i+4], max_player) for i in range(3))

    diagonals = get_diagonals(board)
    for diagonal in diagonals:
        score += np.sum(evaluate_line(diagonal[i:i+4], max_player) for i in range(len(diagonal) - 3))

    return score
```

1.3 Move Ordering:

To improve the efficiency of alpha-beta pruning, the AI orders potential moves based on their distance from the center column. The rationale is that the center columns provide more opportunities for creating winning combinations and blocking the opponent.

```python
def ordering_moves(board):
    possible_moves = board.get_possible_moves()
    unfilled_moves = [move for move in possible_moves if not board.column_filled(move)]

    center_column = board.grid.shape[1] // 2
    unfilled_moves.sort(key=lambda move: abs(move - center_column))

    return unfilled_moves
```

2. Minimax Algorithm with Alpha-Beta Pruning

The alpha-beta pruning algorithm is employed to efficiently search through the game tree and avoid unnecessary calculations. It minimizes the number of nodes evaluated in the minimax tree by maintaining bounds on the possible scores. The evaluation function helps in determining when to prune branches by providing a score for each game state.

2.1 Maximize Value

The max_value function aims to find the maximum achievable value for a given game state. It evaluates potential moves by exploring the game tree, considering both immediate gains and long-

term consequences. The function employs the alpha-beta pruning mechanism to discard unproductive branches, ensuring a more efficient search.

```
def max_value(board, depth, alpha, beta, max_player):
    if depth == 0 or board.check_victory():
        return evaluate_position(board, max_player)

    value = -inf
    best_move = None

    for move in ordering_moves(board):
        if not board.column_filled(move):
            child_board = board.copy()
            child_board.add_disk(move, max_player, update_display=False)
            child_value = min_value(child_board, depth - 1, alpha, beta, max_player)
            value = max(value, child_value)

            if value >= beta: return value
            alpha = max(alpha, value)

            if best_move is None or child_value > value:
                best_move = move
    return value
```

2.2 Minimize Value

The min_value function, conversely, seeks to minimize the value associated with potential opponent moves. It follows a similar recursive exploration of the game tree, evaluating each possible move's consequences. The function employs alpha-beta pruning to eliminate branches that cannot lead to a better outcome for the opponent.

```
def min_value(board, depth, alpha, beta, max_player):
    if depth == 0 or board.check_victory():
        return evaluate_position(board, max_player)

    value = inf
    best_move = None
    for move in ordering_moves(board):
        if not board.column_filled(move):
            child_board = board.copy()
            child_board.add_disk(move, 3 - max_player, update_display=False)
            child_value = max_value(child_board, depth - 1, alpha, beta, max_player)
            value = min(value, child_value)

            if value <= alpha: return value
            beta = min(beta, value)

            if best_move is None or child_value < value:
                best_move = move
    return value
```

3. Conclusion:

The combination of the evaluation function, move ordering, and alpha-beta pruning allows the AI to make strategic decisions in Connect Four efficiently. The evaluation function provides a nuanced understanding of the game state, guiding the AI towards moves that increase the likelihood of winning while preventing the opponent from doing so. The alpha-beta pruning algorithm ensures that the AI explores the game tree effectively, making the Connect Four AI both intelligent and computationally efficient.