

Implementation Report – Connect Four with Minimax AI

1. Game Mechanics Implementation:

- Board Representation:

The game board is represented as a 6 x 7 NumPy matrix:

- 0 → empty cell
- 1 → human player
- 2 → AI player

```
*def create_board():    # This function initializes an empty board.  
    return np.zeros((row_count, column_count), int)
```

- Placing Pieces:

```
* def is_valid_location(board, col):    # Checks if a column is not full.  
    return board[0][col] == 0
```

```
*def get_next_open_row(board, col):    # Finds the lowest available row in a  
column.
```

```
for r in range(row_count - 1, -1, -1):  
    if board[r][col] == 0:  
        return r
```

```
*def drop_piece(board, row, col, piece):    # Places a piece on the board.  
    board[row][col] = piece
```

- Winning Condition Check:

```
* def winning_state(board, piece):
```

-This function checks whether the given player has achieved 4 connected pieces. The check is done in four directions:

- Horizontal
- Vertical
- Positive diagonal (\)
- Negative diagonal (/)

If any sequence of four identical pieces is found, the function returns True.

- Valid Columns:

```
* def get_valid_locations(board):
    return [col for col in range(column_count) if is_valid_location(board, col)]
```

Returns a list of playable columns (not full).

2. AI Implementation (Minimax with Alpha-Beta Pruning)

-Heuristic Evaluation:

```
* def evaluate_window(window, piece):
```

The AI evaluates every 4-cell window on the board:

- 4 AI pieces → +100 points
- 3 AI pieces + 1 empty → +5
- 2 AI pieces + 2 empty → +2
- 3 opponent pieces + 1 empty → -4 (block opponent)

This helps the AI both attack and defend.

```
*def score_position(board, piece):
```

Scores the whole board by scanning all rows, columns, and diagonals.

The center column is rewarded because it leads to more possible winning lines.

- Terminal State Check:

```
* def is_terminal_node(board):
```

Returns True if:

- The AI wins
- The player wins
- The board is full (draw)

This stops the minimax recursion.

- Minimax + Alpha-Beta Pruning:

```
* def minimax(board, depth, alpha, beta, maximizingPlayer):
```

- Recursively simulates future moves up to a given depth.
- `maximizingPlayer=True` → AI's turn (tries to maximize score)
- `maximizingPlayer=False` → Player's turn (tries to minimize score)
- Alpha-beta pruning eliminates branches that cannot influence the result, making the AI faster.

The function returns:

- The best column to play
- The evaluated score of that move

-Selecting the Best Move:

```
*def get_best_move(board):
    column, _ = minimax(...)
    return column
```

Uses the minimax result to choose the optimal move.

3. Main Game Loop:

```
* def main():
```

Creates the board

Alternates turns between player and AI

After each move, prints the updated board

If a win or draw is detected, the game ends

The human player selects a column via input, while the AI automatically selects one using minimax.

-Conclusion

This project implements a fully working Connect Four game with an AI opponent. The AI applies a minimax algorithm combined with alpha-beta pruning to simulate future moves and choose the optimal one. The heuristic evaluation allows the AI to both build winning opportunities and block the opponent.

As a result, the AI plays strategically and becomes stronger as the search depth increases.