Seven Six Puzzle Report

Worked In Pair
- Chrysis Andreou (UC1366020)
- Mohamad Fatfat (UC1367680)

Implemented With Alpha-Beta Pruning

1.) Overview of play_game

The `play_game()` function is the core of the game that allows a human player (Player O) to compete against the computer (Player X) in a Seven Six Puzzle. Here's how the function operates:

1. **Board Setup and User Configuration**:
   - It begins by initializing an empty game board and then asks the human player to specify two things: the depth of the minimax algorithm (which controls how "deep" the computer's decision-making goes) and whether they want to enable alpha-beta pruning (a technique that can make the computer's decision-making faster).
2. **Gameplay Loop**:
   - The game alternates turns between the computer and the human player. Player X (the computer) always goes first. On each turn, the computer uses either the `minimax` function (with or without pruning) to decide its move based on the current board state, while the human player chooses a column for their move.
3. **Move Execution and Display**:
   - After each move, the function prints the current board state, showing both the computer's and the human's moves.
4. **Checking for Win or Draw**:
   - After every move, it checks whether a player has won using the `winning_move()` function. If so, the game ends and declares the winner. It also checks if the board is full, resulting in a draw.
5. **Human Interaction**:
   - The human player is prompted to select a column for their move or quit the game at any time.

The function runs in a loop until either a player wins, the game ends in a draw, or the human decides to quit.

2.) Core Game Mechanics

1. Board Setup and Initialization

- **Constants for the game**:
  - Defines the game dimensions, symbols for players, and empty spaces. `ROWS = 7` and `COLS = 6` represent the board size. `EMPTY` is a placeholder for unused cells. `PLAYER_X` is the computer's symbol, while `PLAYER_O` is the human player's symbol.
- **`create_board()`**:
  - Initializes and returns a 7x6 board filled with empty spaces (represented by `EMPTY`). This board is used throughout the game to track the state of the game.

2. Displaying the Game State

- `print_board(board)`:
  - Prints the current state of the game board. It loops through each row of the board and prints it, visually representing the game's current configuration after each move.

3. Move Validation and Location Checks

- `is_valid_location(board, col)`:
  - Checks if a move in a specific column is valid by verifying if the top row of that column is still empty (i.e., the column is not full). This function is essential to ensure that a move can be legally placed in the selected column.
- `get_valid_locations(board)`:
  - Returns a list of columns that are still available for a move, based on whether their top rows are empty. It allows the game to know which columns are still open for both players to play in.

4. Handling Moves on the Board

- `get_next_open_row(board, col)`:
  - Given a column, this function finds the next available row (from the bottom up) where a player's piece can be dropped. It helps the game simulate the "gravity" effect, where pieces fill up from the bottom of the column.
- `drop_piece(board, row, col, piece)`:
  - Places the player's piece (either `PLAYER_X` or `PLAYER_O`) on the board at the specified row and column. This function updates the board to reflect the player's move.

5. Checking for a Win

- `winning_move(board, piece)`:
  - Checks if the specified player (`piece`) has won the game by forming a sequence of four consecutive pieces. It examines:
    - **Horizontal**: Checks each row to see if there are four consecutive pieces of the same type.
    - **Vertical**: Checks each column for vertical sequences of four.
    - **Positively sloped diagonal**: Checks diagonals sloping upwards (from bottom-left to top-right).
    - **Negatively sloped diagonal**: Checks diagonals sloping downwards (from top-left to bottom-right).
  - If any of these conditions are met, it returns `True`, indicating a winning move.

6. Game Termination Check

- `is_terminal_node(board)`:
    - Determines whether the game is over by checking if either player has won (using `winning_move()`) or if there are no valid moves left (i.e., the game is a draw). This function is essential for deciding when the game should end.

3.) Evaluation Functions

In the context of the game, the evaluation functions help assess how favorable a particular board configuration is for a given player. These evaluations are essential for decision-making algorithms like minimax, as they allow the computer to assign a score to each potential move. Here's how the evaluation functions work:

`evaluate_window(window, piece)`

This function evaluates a specific group of four consecutive cells (called a "window") on the game board. It determines how valuable the window is for a particular player (either Player X or Player O) and assigns a score based on certain conditions:

- **If the window contains four of the player's pieces**, the score increases significantly (`+100`), indicating a winning move.
- **If the window contains three of the player's pieces and one empty cell**, the score is moderately increased (`+5`), indicating a strong position where a win could be achieved soon.
- **If the window contains two of the player's pieces and two empty cells**, a smaller score boost (`+2`) is awarded to reflect a potentially advantageous situation.
- **If the opponent has three pieces and one empty cell**, the score is reduced (`-4`), as this represents a dangerous position where the opponent could win next turn. This acts as a defensive penalty.

This function helps the computer prioritize winning moves and block the opponent from winning.

`score_position(board, piece)`

This function calculates the overall score of the entire board for a given player. It combines multiple windows in different directions (horizontal, vertical, and diagonal) and sums up the evaluations to reflect how favorable the board is:

- **Center Column Focus**: The function gives extra weight to placing pieces in the center column (`COLS//2`) because playing centrally allows for more winning opportunities (both horizontally and diagonally). For each piece in the center column, the score is increased (`center_count * 3`).
- **Horizontal Scoring**: It evaluates every possible set of four consecutive cells horizontally and uses `evaluate_window()` to score each window.
- **Vertical Scoring**: Similarly, the function evaluates vertical windows (four consecutive cells in the same column) and applies the same scoring criteria.

- **Diagonal Scoring**: Both positively sloped and negatively sloped diagonal windows are evaluated using the same method.

The result of this function is a total score for the board, which helps the computer decide the best possible move based on the configuration that maximizes its own score while minimizing the opponent's advantage.

4.) Computer's turn

4.1) Overview of computer's turn

In the computer's turn, the AI (Player X) makes its move as follows:

1. **Check Turn**:
    - The game checks if it's the computer's turn, which is identified by `turn == 0`. The computer is the "MAX player" in this scenario, meaning it aims to maximize its score during decision-making.
2. **Use Minimax Algorithm**:
    - The computer uses the minimax algorithm to determine its next move. If alpha-beta pruning is enabled (`use_alpha_beta` is true), the function calls the `minimax()` method, which optimizes the decision-making process by skipping unnecessary branches in the search tree. Otherwise, the computer calls `minimax_no_pruning()`, which explores all possibilities without skipping.
    - Both functions return the best column (`col`) for the computer to place its piece and the associated score (`minimax_score`) for that column based on the current board state.
3. **Check Validity of Move**:
    - Once the best column is determined, the game verifies whether the column is valid by checking if it's not full using the `is_valid_location()` function. If it's valid, the function calculates the next available row in that column using `get_next_open_row()`.
4. **Make the Move**:
    - The computer places its piece (Player X) in the chosen column and row by calling `drop_piece()`.
5. **Display the Move**:
    - After making the move, the computer announces its move by printing the updated game board using `print_board()`.
6. **Check for a Win**:
    - After making the move, the computer checks if it has won by calling `winning_move()`. If the computer wins, the game announces the final configuration of the board, declares the computer as the winner, and ends the game by setting `game_over = True`.

This process is repeated until either the game ends (due to a win or draw) or the computer completes its move successfully without ending the game.

4.2) Minimax algorithm

The **minimax algorithm** with alpha-beta pruning is a decision-making algorithm used in two-player games like the Seven Six Puzzle. It helps determine the best possible move for the computer (Player X) by simulating all possible future moves up to a certain depth. Here’s a breakdown of how it works:

Core Concepts:

1. **Maximizing vs. Minimizing Player**:
    - **Maximizing Player (Player X)**: The computer aims to **maximize** the score (make the best possible move for itself).
    - **Minimizing Player (Player O)**: The human opponent aims to **minimize** the score (make the worst possible move for the computer).
2. **Recursion and Depth**:
    - The algorithm works by recursively evaluating all possible future moves up to a certain "depth." Each recursive call alternates between maximizing and minimizing the score, simulating the two players' actions.
3. **Alpha-Beta Pruning**:
    - This optimization technique is used to "prune" or eliminate unnecessary branches in the decision tree, speeding up the process without affecting the final result. It avoids exploring moves that are clearly worse than others by using two values:
        - **Alpha**: The best value that the maximizing player can guarantee.
        - **Beta**: The best value that the minimizing player can guarantee.

Breakdown of the Code:

1. **Base Case (Depth 0 or Terminal Node)**:
    - The algorithm stops when either:
        - The game reaches a terminal state (a win, loss, or draw).
        - The depth limit is reached.
    - In these cases, the function returns a score that evaluates the board using the `score_position()` function, where:
        - A high score favors Player X (the computer).
        - A low score favors Player O (the human).
        - If a player has won, the score is extremely high or low (depending on which player won).
2. **Maximizing Player’s Turn (Computer)**:
    - The algorithm goes through all valid moves (columns), simulates dropping a piece in each one, and recursively calls itself to evaluate the next move by the minimizing player (Player O).
    - It picks the move that **maximizes** the score. During this process:
        - **Alpha** is updated with the highest value found so far (best move for Player X).

        - If **Alpha** becomes greater than or equal to **Beta**, the function "prunes" the remaining branches (further moves in that column) because the minimizing player (Player O) will avoid this branch. This improves efficiency by skipping unnecessary evaluations.
3. **Minimizing Player’s Turn (Human)**:
    - Similar to the maximizing player’s turn, but now the function looks for moves that **minimize** the score for Player O.
    - **Beta** is updated with the lowest value found so far, and if **Alpha** exceeds **Beta**, the function prunes the remaining branches.
4. **Return Values**:
    - For each recursive call, the function returns:
        - The **best column** for the current player (either Player X or Player O).
        - The **score** of that column based on the minimax evaluation.

Alpha-Beta Pruning Example:

Imagine Player X has a set of moves that guarantee a high score. As soon as Player O finds a move that would result in a lower score (or at least not as bad as others), Player O would choose that move and disregard other options. The algorithm uses **alpha** and **beta** to avoid exploring these unnecessary paths, improving efficiency without compromising accuracy.

In summary, the minimax algorithm evaluates potential game moves to find the best strategy by simulating future moves and using alpha-beta pruning to speed up the process. It alternates between maximizing and minimizing the score while pruning suboptimal paths to find the optimal move for Player X (the computer).

5) Human's turn

In the human's turn (MIN player), the game waits for the human player to make a move by selecting a column or quitting the game. Here’s a breakdown of how this part of the game works:

1. **Prompting for Input**:
    - The human player is prompted to select a column number between 1 and 6 for their next move. They can also type 'Q' or 'q' to quit the game if they want to stop playing.
2. **Handling Quit Option**:
    - If the player enters 'Q' or 'q', the game sets `game_over` to `True` and prints a message indicating that the game has been quit. This exits the loop, ending the game.
3. **Validating Input**:
    - If the player enters a number, the input is first checked to see if it's within the valid range of columns (1 to 6).
    - If the column number is valid (i.e., it corresponds to a column on the board), the game then checks if the column is available for a move using the `is_valid_location()` function, which ensures that the column is not already full.
4. **Executing the Move**:

- If the selected column is valid and has space, the game finds the next open row in that column using the `get_next_open_row()` function. Then, it places the human player's piece ('O') in that row using the `drop_piece()` function.

5. **Displaying the Move**:
   - After the move is made, the updated board is printed so the player can see their piece placed on the board.
6. **Checking for Win**:
   - Once the player makes a move, the game checks if the player has won by forming a four-piece sequence using the `winning_move()` function. If a win is detected, it prints the final board configuration and declares the human player as the winner, setting `game_over` to `True`.
7. **Handling Invalid Input**:
   - If the player selects a column that is full, they are prompted to choose another column. If they input an invalid number (outside the 1-6 range) or something that's not a number, they are asked to input a valid column number or press 'Q' to quit.

This loop continues until the player either successfully makes a valid move, quits, or wins the game.

6) Time performance with and without pruning

The `experiment_game()` function is designed to compare the performance of the minimax algorithm with and without alpha-beta pruning by simulating a game between two players. Here's an explanation of its behavior:

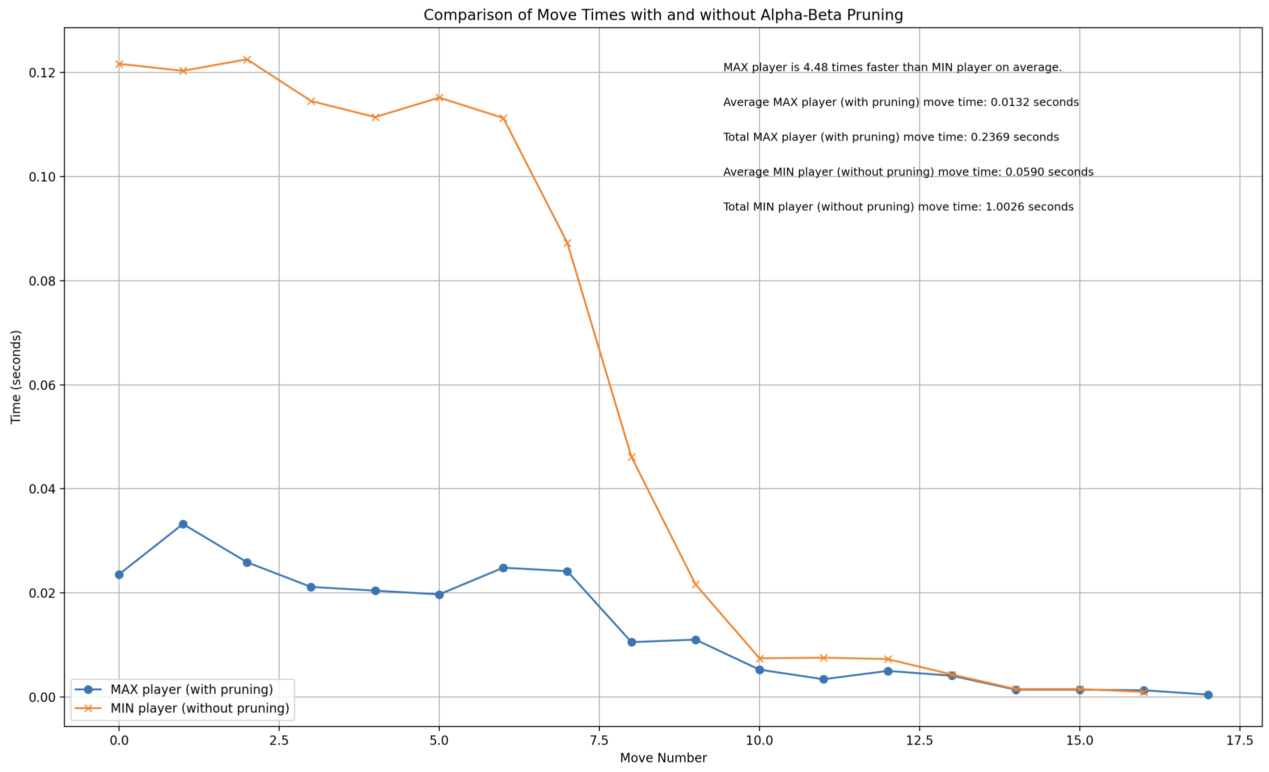
1. **Setup and Initialization**:
   - The game board is initialized, and a list of time measurements (`max_times` and `min_times`) is created to store how long each move takes for the computer.
   - The game alternates turns between two players:
     - **MAX player (Computer)** using the `minimax` algorithm with alpha-beta pruning.
     - **MIN player (Computer)** using the `minimax_no_pruning` algorithm, which does not use alpha-beta pruning.
2. **Gameplay Loop**:
   - The game continues in a loop until one player wins or the game ends in a draw.
   - On each turn, the function records the time taken for that move:
     - **MAX player's turn (alpha-beta pruning)**: The function measures the time taken to compute a move using the `minimax` algorithm with pruning.
     - **MIN player's turn (no pruning)**: Similarly, it measures the time for the `minimax_no_pruning` algorithm.
   - The selected move is then applied to the board, and it checks for a winning move or a draw after each turn.
3. **Time Analysis**:

- After the game ends, the function calculates the **average** and **total** time spent by both the MAX and MIN players.
- It also computes the **speedup ratio**, comparing the average time taken by the MAX player (with pruning) to the MIN player (without pruning). This shows how much faster the alpha-beta pruning makes the algorithm.

4. **Plotting and Visualization**:
   - A plot is generated to visually compare the time taken by each player on every move, showing:
     - The times for the MAX player (with pruning) and MIN player (without pruning) over the course of the game.
   - Text annotations are added to the plot, providing insights into the average times, total times, and the speedup factor.

The function essentially runs a simulation of a full game to demonstrate the performance improvements of using alpha-beta pruning in the minimax algorithm, providing both numerical and visual results.

7.) Performance visualisation



8.) Python code

```
import math

import random
```

```python
import time

import matplotlib.pyplot as plt


# Constants for the game

ROWS = 7

COLS = 6

EMPTY = '.'

PLAYER_X = 'X' # Computer (MAX player)

PLAYER_O = 'O' # Human (MIN player)


# Initialize the game board with empty spaces

def create_board():

return [[EMPTY for _ in range(COLS)] for _ in range(ROWS)]


# Print the current state of the board

def print_board(board):

for row in board:

print("".join(row))

print()


# Check if the top row of the column is empty, indicating a valid move

def is_valid_location(board, col):

return board[0][col] == EMPTY
```

```
# Return a list of columns that are not full

def get_valid_locations(board):

valid_locations = []

for col in range(COLS):

if is_valid_location(board, col):

valid_locations.append(col)

return valid_locations



# The user input is a column number, we need to find the next open row in that
column

def get_next_open_row(board, col):

for row in range(ROWS-1, -1, -1): # from last row to top row

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

return row



# Place the player's piece in the specified location

def drop_piece(board, row, col, piece):

board[row][col] = piece



# Check for a winning move in all directions: horizontal, vertical, and diagonal

def winning_move(board, piece):

# Horizontal check

for row in range(ROWS):

for col in range(COLS-3):

if all(board[row][col+i] == piece for i in range(4)):
```

```
return True


# Check vertical locations for win

for row in range(ROWS-3):

for col in range(COLS):

if all(board[row+i][col] == piece for i in range(4)):

return True


# Check positively sloped diagonals

for row in range(ROWS-3):

for col in range(COLS-3):

if all(board[row+i][col+i] == piece for i in range(4)):

return True


# Check negatively sloped diagonals

for row in range(3, ROWS):

for col in range(COLS-3):

if all(board[row-i][col+i] == piece for i in range(4)):

return True


return False


# Determine if the game is over (win or draw)

def is_terminal_node(board):

return winning_move(board, PLAYER_X) or winning_move(board, PLAYER_O) or
```

```
len(get_valid_locations(board)) == 0


# Evaluate a window of four cells for scoring

def evaluate_window(window, piece):

score = 0

opp_piece = PLAYER_O if piece == PLAYER_X else PLAYER_X


# Scoring based on the number of pieces in the window

if window.count(piece) == 4:

score += 100

elif window.count(piece) == 3 and window.count(EMPTY) == 1:

score += 5

elif window.count(piece) == 2 and window.count(EMPTY) == 2:

score += 2


# Penalize if the opponent is close to winning

if window.count(opp_piece) == 3 and window.count(EMPTY) == 1:

score -= 4


return score


# Calculate the score of the board for the given piece

def score_position(board, piece):

score = 0
```

```
# Score center column to encourage central play to allow for more opportunities
for a win

center_array = [board[i][COLS//2] for i in range(ROWS)] # COLS//2 is the center
column

center_count = center_array.count(piece)

score += center_count * 3



# Score Horizontal

for row in range(ROWS):

row_array = [board[row][i] for i in range(COLS)]

for col in range(COLS-3):

window = row_array[col:col+4]

score += evaluate_window(window, piece)



# Score Vertical

for col in range(COLS):

col_array = [board[row][col] for row in range(ROWS)]

for row in range(ROWS-3):

window = col_array[row:row+4]

score += evaluate_window(window, piece)



# Score positive sloped diagonal

for row in range(ROWS-3):

for col in range(COLS-3):

window = [board[row+i][col+i] for i in range(4)]
```

```
score += evaluate_window(window, piece)


# Score negative sloped diagonal

for row in range(3, ROWS):

for col in range(COLS-3):

window = [board[row-i][col+i] for i in range(4)]

score += evaluate_window(window, piece)


return score


# Minimax algorithm without alpha-beta pruning

def minimax_no_pruning(board, depth, maximizingPlayer):

valid_locations = get_valid_locations(board)

is_terminal = is_terminal_node(board)

if depth == 0 or is_terminal:

if is_terminal:

if winning_move(board, PLAYER_X):

return (None, 100000000000000) # None indicates no specific column move as the
game is over

elif winning_move(board, PLAYER_O):

return (None, -10000000000000)

else: # Game is over,draw, no more valid moves

return (None, 0)

else: # Depth is zero

return (None, score_position(board, PLAYER_X))
```

```
if maximizingPlayer:

value = -math.inf # Initialize value to negative infinity, to ensure that any
score calculated will be higher than this initial value

best_col = random.choice(valid_locations) # serves as a default choice in case no
better column is found.

for col in valid_locations:

row = get_next_open_row(board, col) # find the next open row in the column

b_copy = [row[:] for row in board] # create a copy of the board

drop_piece(b_copy, row, col, PLAYER_X) # drop the piece in the copy of the board

new_score = minimax_no_pruning(b_copy, depth-1, False)[1] # recursively call
minimax_no_pruning to evaluate the score of the new board state

if new_score > value:

value = new_score

best_col = col

return best_col, value



else: # Minimizing player

value = math.inf

best_col = random.choice(valid_locations)

for col in valid_locations:

row = get_next_open_row(board, col)

b_copy = [row[:] for row in board]

drop_piece(b_copy, row, col, PLAYER_O)

new_score = minimax_no_pruning(b_copy, depth-1, True)[1] # now call with True to
indicate that the next call is the MAX player's turn

if new_score < value:

value = new_score
```

```
best_col = col

return best_col, value



# Minimax algorithm with alpha-beta pruning to find the best move

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

valid_locations = get_valid_locations(board)

is_terminal = is_terminal_node(board)

if depth == 0 or is_terminal:

if is_terminal:

if winning_move(board, PLAYER_X):

return (None, 100000000000000)

elif winning_move(board, PLAYER_O):

return (None, -10000000000000)

else: # Game is over, no more valid moves

return (None, 0)

else: # Depth is zero

return (None, score_position(board, PLAYER_X))

if maximizingPlayer:

value = -math.inf

best_col = random.choice(valid_locations)

for col in valid_locations:

row = get_next_open_row(board, col)

b_copy = [row[:] for row in board]

drop_piece(b_copy, row, col, PLAYER_X)
```

```
new_score = minimax(b_copy, depth-1, alpha, beta, False)[1] # [1] is the score of
the new board state

if new_score > value:

value = new_score

best_col = col



'''If alpha is greater than or equal to beta,

the minimizing player will avoid this branch because they have a better option
elsewhere in the tree.

Therefore, further exploration of this branch is unnecessary,

and the algorithm can "prune" it, skipping the evaluation of any further nodes in
this branch.'''

alpha = max(alpha, value)

if alpha >= beta:

break

return best_col, value



else: # Minimizing player

value = math.inf

best_col = random.choice(valid_locations)

for col in valid_locations:

row = get_next_open_row(board, col)

b_copy = [row[:] for row in board]

drop_piece(b_copy, row, col, PLAYER_O)

new_score = minimax(b_copy, depth-1, alpha, beta, True)[1]

if new_score < value:
```

```
value = new_score

best_col = col

# Update beta to the minimum value found so far to ensure the minimizing player
chooses the least score possible for MAX

beta = min(beta, value)

if alpha >= beta:

break

return best_col, value



# Main function to play the game

def play_game():

board = create_board()

game_over = False

turn = 0 # Computer starts first



print("Let us play Seven Six Puzzle. I am X and you are O.")



# Ask the user for the depth of the minimax algorithm

while True:

try:

depth = int(input("Enter the depth for the minimax algorithm between 1 and 6: "))

if 1 <= depth <= 6:

break

else:

print("Please enter a depth between 1 and 6.")
```

```
except ValueError:

print("Invalid input. Please enter a number between 1 and 6.")



# Ask the user if they want to use alpha-beta pruning

while True:

use_alpha_beta_input = input("Do you want to use alpha-beta pruning? (y/n): 
").strip().lower()

if use_alpha_beta_input in ['y', 'n']:

use_alpha_beta = use_alpha_beta_input == 'y' # True if y, False if n

break

else:

print("Invalid input. Please enter 'y' or 'n'.")



# Print the initial board

print("This is the board")

print_board(board)

print("and I play first")



while not game_over:

if turn == 0:

# Computer's turn (MAX player)

if use_alpha_beta:

col, minimax_score = minimax(board, depth, -math.inf, math.inf, True)

else:

col, minimax_score = minimax_no_pruning(board, depth, True)
```

```
if is_valid_location(board, col):

row = get_next_open_row(board, col)

drop_piece(board, row, col, PLAYER_X)



print("This is my move")

print_board(board)



if winning_move(board, PLAYER_X):

print("The final configuration")

print_board(board) # Reprint the final configuration

print(">>> I am the Winner! <<<")

game_over = True



else:

# Human's turn (MIN player)

while True:

user_input = input("Select the column for your next move (1-6) or press 'Q' to
quit: ")

if user_input.lower() == 'q':

print(">>> Game has been quit. Goodbye! <<<")

game_over = True

break



if user_input.isdigit():

col = int(user_input) - 1
```

```
if 0 <= col < COLS:

if is_valid_location(board, col):

row = get_next_open_row(board, col)

drop_piece(board, row, col, PLAYER_O)



print("This is what you played")

print_board(board)



if winning_move(board, PLAYER_O):

print("The final configuration")

print_board(board) # Reprint the final configuration

print(">>> You are the Winner! <<<")

game_over = True

break

else:

print("Column is full. Please choose another column.")

else:

print("Invalid column. Please enter a number between 1 and 6.")

else:

print("Invalid input. Please enter a number between 1 and 6 or 'Q' to quit.")



# Check for a draw

if not game_over and len(get_valid_locations(board)) == 0:

print(">>> The game is a draw! <<<")

game_over = True
```

```
turn += 1

turn = turn % 2 # Alternates between 0 and 1


if __name__ == "__main__":

play_game()


def experiment_game(depth=4):

max_times = []

min_times = []

experiment_starts = []


board = create_board()

game_over = False

turn = 0 # MAX player starts first


# Record the start index of the experiment

experiment_starts.append(len(max_times))


while not game_over:

if turn == 0:

# MAX player's turn (with alpha-beta pruning)

start_time = time.time()

col, minimax_score = minimax(board, depth, -math.inf, math.inf, True)
```

```
end_time = time.time()

max_times.append(end_time - start_time)



if is_valid_location(board, col):

row = get_next_open_row(board, col)

drop_piece(board, row, col, PLAYER_X)



if winning_move(board, PLAYER_X):

game_over = True



else:

# MIN player's turn (without alpha-beta pruning)

start_time = time.time()

col, minimax_score = minimax_no_pruning(board, depth, False)

end_time = time.time()

min_times.append(end_time - start_time)



if is_valid_location(board, col):

row = get_next_open_row(board, col)

drop_piece(board, row, col, PLAYER_O)



if winning_move(board, PLAYER_O):

game_over = True



# Check for a draw
```

```
if not game_over and len(get_valid_locations(board)) == 0:

game_over = True



turn += 1

turn = turn % 2 # Alternates between 0 and 1



# Calculate average and total times

avg_max_time = sum(max_times) / len(max_times)

total_max_time = sum(max_times)

avg_min_time = sum(min_times) / len(min_times)

total_min_time = sum(min_times)



# Calculate how much faster MAX is compared to MIN

if avg_min_time > 0:

speedup_ratio = avg_min_time / avg_max_time

speedup_text = f"MAX player is {speedup_ratio:.2f} times faster than MIN player
on average."

else:

speedup_text = "MIN player did not make any moves, so speed comparison is not
applicable."



avg_max_text = f"Average MAX player (with pruning) move time: {avg_max_time:.4f}
seconds"

total_max_text = f"Total MAX player (with pruning) move time:
{total_max_time:.4f} seconds"

avg_min_text = f"Average MIN player (without pruning) move time:
{avg_min_time:.4f} seconds"
```

```
total_min_text = f"Total MIN player (without pruning) move time:
{total_min_time:.4f} seconds"



# Plot the times

plt.figure(figsize=(10, 5))

plt.plot(max_times, label='MAX player (with pruning)', marker='o')

plt.plot(min_times, label='MIN player (without pruning)', marker='x')



plt.xlabel('Move Number')

plt.ylabel('Time (seconds)')

plt.title('Comparison of Move Times with and without Alpha-Beta Pruning')

plt.legend()

plt.grid(True)



# Add text annotations to the plot

plt.text(0.55, 0.95, speedup_text, transform=plt.gca().transAxes, fontsize=9,
verticalalignment='top')

plt.text(0.55, 0.90, avg_max_text, transform=plt.gca().transAxes, fontsize=9,
verticalalignment='top')

plt.text(0.55, 0.85, total_max_text, transform=plt.gca().transAxes, fontsize=9,
verticalalignment='top')

plt.text(0.55, 0.80, avg_min_text, transform=plt.gca().transAxes, fontsize=9,
verticalalignment='top')

plt.text(0.55, 0.75, total_min_text, transform=plt.gca().transAxes, fontsize=9,
verticalalignment='top')



plt.show()
```

```
# Run the experiment

experiment_game()
```

9.) Code Output

~~~
```Let us play Seven Six Puzzle. I am X and you are O.
Enter the depth for the minimax algorithm between 1 and 6: 5
Do you want to use alpha-beta pruning? (y/n): y
This is the board
......
......
......
......
......
......
......

and I play first
This is my move
......
......
......
......
......
......
...X..

Select the column for your next move (1-6) or press 'Q' to quit: 4
This is what you played
......
......
......
......
......
...O..
...X..

This is my move
......
......
......
......
...X..
...O..
...X..
~~~

```
Select the column for your next move (1-6) or press 'Q' to quit: 4
This is what you played
......
......
......
...O..
...X..
...O..
...X..

This is my move
......
......
......
...O..
...X..
...O..
.X.X..

Select the column for your next move (1-6) or press 'Q' to quit: 3
This is what you played
......
......
......
...O..
...X..
...O..
.XOX..

This is my move
......
......
......
...O..
...X..
..XO..
.XOX..

Select the column for your next move (1-6) or press 'Q' to quit: 3
This is what you played
......
......
......
...O..
..OX..
..XO..
.XOX..
```

```
This is my move
......
......
......
...O..
..OX..
..XO..
XXOX..

Select the column for your next move (1-6) or press 'Q' to quit: 2
This is what you played
......
......
......
...O..
..OX..
.OXO..
XXOX..

This is my move
......
......
......
..XO..
..OX..
.OXO..
XXOX..

Select the column for your next move (1-6) or press 'Q' to quit: 2
This is what you played
......
......
......
..XO..
.OOX..
.OXO..
XXOX..

This is my move
......
......
......
.XXO..
.OOX..
.OXO..
XXOX..

Select the column for your next move (1-6) or press 'Q' to quit: 3
This is what you played
```

```
......
......
..O...
.XXO..
.OOX..
.OXO..
XXOX..

This is my move
......
......
.XO...
.XXO..
.OOX..
.OXO..
XXOX..

Select the column for your next move (1-6) or press 'Q' to quit: 2
This is what you played
......
.O....
.XO...
.XXO..
.OOX..
.OXO..
XXOX..

This is my move
......
.O....
.XOX..
.XXO..
.OOX..
.OXO..
XXOX..

Select the column for your next move (1-6) or press 'Q' to quit: 3
This is what you played
......
.OO...
.XOX..
.XXO..
.OOX..
.OXO..
XXOX..

This is my move
......
.OOX..
```

```
.XOX..
.XXO..
.OOX..
.OXO..
XXOX..

Select the column for your next move (1-6) or press 'Q' to quit: 4
This is what you played
...O..
.OOX..
.XOX..
.XXO..
.OOX..
.OXO..
XXOX..

This is my move
..XO..
.OOX..
.XOX..
.XXO..
.OOX..
.OXO..
XXOX..

Select the column for your next move (1-6) or press 'Q' to quit: 2
This is what you played
.OXO..
.OOX..
.XOX..
.XXO..
.OOX..
.OXO..
XXOX..

This is my move
.OXO..
.OOX..
.XOX..
.XXO..
.OOX..
XOXO..
XXOX..

Select the column for your next move (1-6) or press 'Q' to quit: 1
This is what you played
.OXO..
.OOX..
.XOX..
```

```
.XXO..
OOOX..
XOXO..
XXOX..

This is my move
.OXO..
.OOX..
.XOX..
.XXO..
OOOX..
XOXO..
XXOX.X

Select the column for your next move (1-6) or press 'Q' to quit: 6
This is what you played
.OXO..
.OOX..
.XOX..
.XXO..
OOOX..
XOXO.O
XXOX.X

This is my move
.OXO..
.OOX..
.XOX..
XXXO..
OOOX..
XOXO.O
XXOX.X

Select the column for your next move (1-6) or press 'Q' to quit: 1
This is what you played
.OXO..
.OOX..
OXOX..
XXXO..
OOOX..
XOXO.O
XXOX.X

This is my move
.OXO..
XOOX..
OXOX..
XXXO..
OOOX..
```

```
XOXO.O
XXOX.X

The final configuration
.OXO..
XOOX..
OXOX..
XXXO..
OOOX..
XOXO.O
XXOX.X

>>> I am the Winner! <<<
```