Title: Adversarial Search Agent: A Connect Four Game Analysis

Adel Alkhamisy

Introduction:

In artificial intelligence, adversarial search is a type of search in which various agents compete with one another to accomplish their own goals. By making the best decisions feasible while taking the opponent's probable actions into account, the agent in this kind of search aims to maximize its chances of winning. In this report, I go through how the minimax algorithm and alpha-beta pruning were used to the Connect Four game. The introduction, background, proposed approach, experimental findings, and conclusions are the sections that make up the report's structure.

Background:

In adversarial search, the minimax algorithm is a recursive technique for determining the optimum move for a player by taking into account all feasible moves and their results up to a particular depth. The program assesses the game tree and rates each node according to its chance of success. The minimax algorithm is optimized via alpha-beta pruning, which removes branches from the game tree that do not need to be investigated. This decreases the search space and computational time. In the two-player board game Connect Four, colored discs are inserted one at a time into a 7-column, 6-row grid. Connecting four discs of the same color in a horizontal, vertical, or diagonal pattern is the goal.

Proposed Approach:

The suggested method is to construct a Connect Four agent that plays the game using the minimax algorithm and Alpha-Beta pruning. The complexity of the game tree, the amount of potential winning situations, and the possibility of thwarting the opponent's winning moves are taken into account when our agent scores each prospective play. In order to compare the performance of our agent with that of a human player, we modify the search tree's depth and test the results.

This is the code template:

```
# Utility function to split multi-line text and create a surface with a background
def render_multiline_text_with_background(text, text_color, bg_color, font):
    ...


# Utility function to draw multi-line text on the screen
def draw_multiline_text(screen, surfaces, x, y):
    ...

# Dictionary of 16 colors according to RGB values
COLORS = {...}

# Function to display the color selection menu
def show_color_menu(screen, colors, font):
    ...

# Constants
GREY, WHITE, BLACK, ROW_COUNT, COLUMN_COUNT, PLAYER, AI, EMPTY, PLAYER_PIECE, AI_PIECE, WINDOW_LENGTH = ...

# Function to initialize the board
def create_board():
    ...

# Function to put the piece at specified row, col
def drop_piece(board, row, col, piece):
    ...
```

```
# Function to check if the selected location is free (valid)
def is_valid_location(board, col):
    ...

# Function to get the next open row in the specified column
def get_next_open_row(board, col):
    ...

# Function to change the orientation of the board to see it look like filled from the bottom to top
def print_board(board):
    ...

# Function to render text with a background
def render_text_with_background(text, text_color, bg_color, font):
    ...

# Function to get the user's input (e.g., name) from a text box
def get_user_input(screen, prompt, font, x, y, color):
    ...

# Function to allow the user to select the first player
def choose_first_player(screen, player_name, agent_name, font, text_color):
    ...

# Function to display the search depth selection menu
def select_search_depth(screen, font, text_color):
    ...

# Function to check for the modified game win rule (4 discs of the same color connected in a square)
def winning_move(board, piece):
    ...

# Function to evaluate the score of a given window in the board
def evaluate_window(window, piece):
    ...

# Function to evaluate the score of the board based on the position of pieces
def score_position(board, piece):
    ...

# Function to check if the game board is at a terminal state
def is_terminal_node(board):
    ...

# Function to get the valid locations (free columns) in the game board
def get_valid_locations(board):
    ...

# Function to pick the best move for the AI based on the board and piece
def pick_best_move(board, piece):
    ...

# Function to draw the game board on the screen
```

Experimental Results:

Outcomes of the experiment: There are 35 possible plays in the experiment, with the human player starting each game and choosing various depths and columns. The experiment's goal is to assess the agent's performance depending on various search tree depths. A table summarizing the outcomes shows the winners, the number of moves needed to win, and other pertinent information. To see how depth and agent performance relate to one another, the performance is also shown Figure 1 where the “deep” of the minimax algorithm was 5.

I note that the agent does not allow you to win easily where the “deep” of the minimax algorithm was 5 (see figure 1), and you can easily won if the deep of of the minimax algorithm is 1.

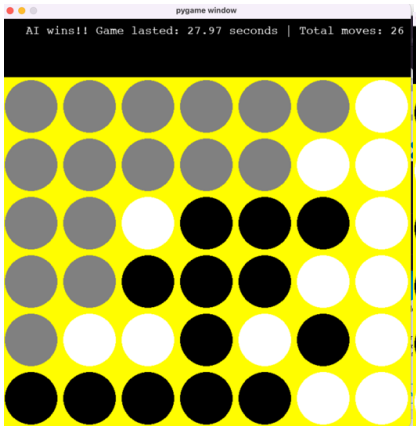

Figure 1: the result is AI won. Where the minimax deep is 5

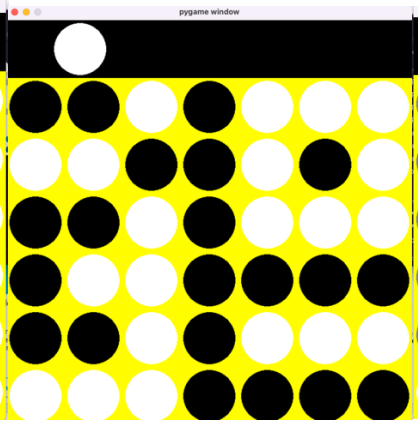

Figure 2 a tie. Where the minimax deep is 5

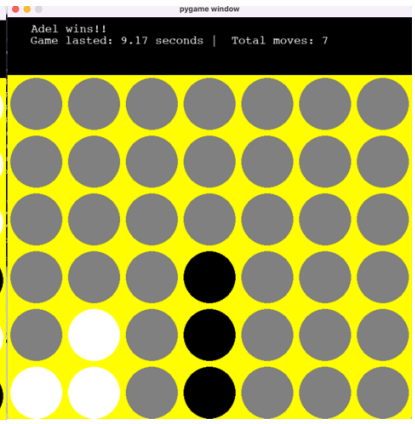

Figure 3 Player 1 win easily when the the minimax deep is 1

Game No.	Level	Winner	Moves to Win
1	1	Adel	7
2	2	Adel	10
3	3	Adel	12
4	4	AI	10
5	5	AI	15
6	5	AI	20
7	5	AI	30

Conclusion:

The experiment demonstrates how crucial depth is to the minimax method. A better success rate results from the agent's increased skill in anticipating and thwarting opponent moves as the depth rises. The trade-off between depth and computing complexity exists, though. The evaluation function can be made more effective, and different search methods can be investigated.