Connect Four Game in Python

Cihan Yildirim
github.com/cihanyildirim1
cihanecmi1@gmail.com

***Abstract*—This paper presents the design and implementation of an AI-powered Connect Four game in Python. The system supports both a human versus AI mode and a traditional two-player mode. The AI employs a minimax algorithm enhanced with alpha-beta pruning to efficiently evaluate game states and select optimal moves. In this document, we describe the overall architecture, key sections of the codebase—including board management, game logic, and graphical interface—and provide a detailed discussion of the minimax algorithm.**

I. Introduction

Connect Four is a widely recognized two-player strategy game where players alternate dropping colored discs into a vertically suspended grid. The primary objective is to be the first to form a horizontal, vertical, or diagonal line of four consecutive pieces. Originally marketed in the 1970s, Connect Four has since become a staple in both casual gaming and academic research in artificial intelligence due to its perfect-information structure and moderate state-space complexity.

In our project, the game is implemented in Python and supports two distinct operational modes:

- **AI Mode:** The computer employs a minimax algorithm enhanced with alpha-beta pruning to evaluate future game states and decide its moves.
- **Two-Player Mode:** Two human players compete by alternately placing their pieces on the board.

The project utilizes NumPy for efficient board representation and Pygame to create an engaging graphical user interface. This combination of technologies not only provides a clear and responsive gameplay experience but also lays the groundwork for exploring more advanced AI strategies in game theory.

II. Background

The minimax algorithm is a cornerstone of decision-making in two-player, zero-sum games and has its origins in early game theory, notably in the work of von Neumann and Morgenstern in the 1940s. The algorithm works by simulating all possible moves for both players, assuming that each player will act optimally to maximize their advantage or minimize their losses. In this context, the algorithm recursively evaluates the game tree, assigning heuristic values to terminal states, and backtracks to determine the optimal move.

One major enhancement to the minimax algorithm is **alpha-beta pruning**, a technique introduced to reduce the number of nodes that need to be evaluated in the game tree. By maintaining two parameters—α (the best already explored option along the path to the root for the maximizer) and β (the best option for the minimizer)—the algorithm can eliminate branches that will not influence the final decision. This pruning significantly increases computational efficiency, allowing for deeper searches within the game tree without a corresponding exponential increase in processing time.

Historically, the combination of minimax and alpha-beta pruning has been instrumental in the development of early computer chess programs and remains a foundational technique in many modern game-playing AIs. While Connect Four is less complex than chess, it still offers a sufficiently rich structure that makes it an ideal test-bed for these algorithms. The balance between computational feasibility and strategic depth makes Connect Four a compelling choice for research in AI, as it allows for exploration of algorithm optimization while providing clear visual feedback through gameplay.

This project not only demonstrates the application of these classic algorithms in a modern programming language but also highlights the enduring relevance of foundational AI concepts in solving strategic game problems.

This work was developed as part of a project on AI and game design.

III. Design and Implementation

The project is structured into several key components:

A. Code Structure Overview

1) **Board Management:** The game board is represented as a 6-by-7 grid using a NumPy array. Functions in this section initialize the board, drop a piece into a column, and check for valid moves.
2) **Game Logic:** This part of the code is responsible for detecting win conditions by scanning the board for four consecutive pieces in horizontal, vertical, or diagonal directions.
3) **Graphical User Interface (GUI):** Pygame is used to render the game board, display game status (e.g., player turns), and handle user input events such as mouse movements and clicks.
4) **AI Decision Making:** The core of the AI is the minimax algorithm with alpha-beta pruning. This section evaluates potential moves by simulating future game states to a specified depth and returns the best move along with its heuristic score.

B. Minimax Algorithm with Alpha-Beta Pruning

The AI component explores the game tree by recursively evaluating all valid moves up to a fixed depth. It then assigns scores to each board state using a heuristic that rewards winning moves and penalizes losing ones.

Algorithm 1 Minimax with Alpha-Beta Pruning for Connect Four

Input: Current board state, search depth, α, β, maximizingPlayer flag
Output: Best move and associated score
if depth is 0 or game over **then**
  return heuristic score of the board
end if
if maximizingPlayer **then**
  Initialize value $\leftarrow -\infty$
  for each valid move **do**
    Simulate move and evaluate resulting board recursively
    Update value $\leftarrow \max(\text{value}, \text{score})$
    Update $\alpha \leftarrow \max(\alpha, \text{value})$
    if $\alpha \geq \beta$ **then**
      break
    end if
  end for
else
  Initialize value $\leftarrow +\infty$
  for each valid move **do**
    Simulate move and evaluate resulting board recursively
    Update value $\leftarrow \min(\text{value}, \text{score})$
    Update $\beta \leftarrow \min(\beta, \text{value})$
    if $\alpha \geq \beta$ **then**
      break
    end if
  end for
end if
return best move and score

C. Discussion of Key Code Sections

- **Board Creation and Management:** The game board is implemented using a two-dimensional NumPy array initialized to zeros. This representation provides an efficient means of storing and manipulating the board state, as NumPy arrays offer fast element-wise operations. Key functions in this section include:
  - `create_board()`: Initializes a 6-by-7 grid where each cell is set to 0, representing an empty slot.
  - `drop_piece(board, row, col, piece)`: Inserts a game piece (e.g., player or AI) into the specified row and column.
  - `get_next_open_row(board, col)`: Scans a column from the bottom up (or vice versa, depending on the indexing) to find the first available row where a piece can be placed.

  This modular approach simplifies board updates and makes the logic for move validation and win detection easier to implement and maintain.
- **Move Validation and Win Detection:** Ensuring that a move is valid is critical for game integrity. This section includes functions that:
  - Verify if a selected column has an empty slot at its topmost position, thus determining whether a new piece can be dropped.
  - Check for a winning move by scanning the board horizontally, vertically, and diagonally. This is achieved by iterating over potential start positions and confirming whether four consecutive positions contain the same non-zero value.

  The win detection logic is optimized to avoid unnecessary checks by limiting iterations to positions where a win is possible, thereby improving overall performance.
- **Graphical Rendering:** The graphical user interface (GUI) is built using the Pygame library, which facilitates the following:
  - Drawing the game board grid with a predefined color scheme and layout.
  - Rendering game pieces with distinct colors (e.g., red for the player, yellow for the AI) so that the board state is clearly visible to the user.
  - Updating the display in real time as the game progresses, including drawing headers and turn indicators.
  - Capturing user input events such as mouse movement and clicks, which are translated into game actions like selecting a column to drop a piece.

  The use of Pygame abstracts much of the low-level graphics handling, allowing the developer to focus on the core game logic while still delivering an engaging visual experience.
- **AI Move Calculation:** The AI component is based on a minimax algorithm enhanced with alpha-beta pruning. This section is central to the AI's decision-making process:
  - `minimax(board, depth, alpha, beta, maximizingPlayer)`: This recursive function evaluates potential moves by simulating future game states to a specified depth. The evaluation function assigns high positive scores to winning states for the AI and high negative scores to states favorable for the opponent.
  - Alpha-beta pruning is employed to cut off branches of the game tree that cannot influence the final decision. By maintaining and updating two parameters, α and β, the algorithm efficiently narrows down the move options, thus reducing computation time without sacrificing decision quality.
  - The function returns both the best move (column index) and its associated heuristic value. This dual output allows the game loop to execute the optimal move while also providing insight into the AI's evaluation strategy.

  The combination of minimax with alpha-beta pruning ensures that the AI remains competitive even as it explores a large number of potential game states.

IV. Conclusion

In this work, we developed an AI-powered Connect Four game in Python that supports both human versus AI and two-player modes. The design leverages a modular structure to separate board management, game logic, GUI, and AI decision-making. The implementation of the minimax algorithm with alpha-beta pruning forms the core of the AI, providing efficient and competitive move selection. Future enhancements could include refining the heuristic evaluation and incorporating additional features such as variable difficulty settings or online multiplayer support.

References

[1] A. Knuth, *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms*, 3rd ed., Addison-Wesley, 1997.

[2] J. Von Neumann and O. Morgenstern, *Theory of Games and Economic Behavior*, Princeton University Press, 1944.

[3] R. E. Korf, "Depth-first Iterative-Deepening: An Optimal Admissible Tree Search," *Artificial Intelligence*, vol. 27, pp. 97–109, 1985.