Connect Four - AI Agent Project

Your Name

February 10th 2024

1 Introduction

In this project, we implemented an AI agent to play the game of Connect Four. The agent utilizes both the alpha-beta search algorithm and the expectimax search algorithm to select the next move given the current board state. The goal of the project is to develop a robust AI player capable of making intelligent decisions and providing a challenging opponent for human players or other AI players.

2 Project Structure

The project consists of two main files:

1. **ConnectFour.py**: This file contains all the functions related to the Connect Four game, including board initialization, making moves, checking for wins or draws, and displaying the current board state. It serves as the backbone of the game logic.
2. **Player.py**: This file contains different types of players that can participate in the game. These include:
    - **AIPlayer**: This is our implementation, where the AI agent is implemented to make moves using the alpha-beta search algorithm and the expectimax search algorithm.
    - **RandomPlayer**: Chooses from valid columns with equal probability.
    - **HumanPlayer**: Represents a human player, allowing interaction with the game through user input.

3 Alpha-Beta Search Algorithm

The alpha-beta search algorithm is a variant of the minimax algorithm used in decision trees. It helps the AI player determine the best move by minimizing the number of nodes that need to be evaluated in the game tree. This optimization

is achieved through pruning, which eliminates unnecessary branches of the tree that will not affect the final decision.

4 Expectimax Search Algorithm

The expectimax search algorithm is another variant of the minimax algorithm that accounts for uncertainty or randomness in games. Unlike the alpha-beta search, which assumes the opponent plays optimally, the expectimax search considers the expected value of chance nodes. This makes it particularly useful in games with elements of chance, such as Connect Four.

5 Conclusion

By implementing both the alpha-beta search algorithm and the expectimax search algorithm, our AI agent is capable of making strategic decisions and providing a challenging gameplay experience. Through rigorous testing and optimization, we aim to create an AI player that can compete with human players and other AI players effectively.

For detailed implementation code and further information, please refer to the `ConnectFour.py` and `Player.py` files.

Miscellaneous Information

Project Implementation

To play the game, you need to run `python ConnectFour.py arg1 arg2`, where `arg1` and `arg2` are one of `AI`, `random`, or `human`.

For example, if you wanted to play against a random player, you would run `python ConnectFour.py human random`.

If you wanted your AI to play itself, you would run `python ConnectFour.py ai ai`. `ConnectFour.py` takes one optional argument `--time` that is an integer. It is the value used to limit the amount of time in seconds to wait for the AI player to make a move. The default value is 60 seconds. **Note:** A human player has to enter their move into the terminal.

Heuristic Used

I implemented the AI agent using the Alpha-beta and minimax heuristic with an evaluation function described below:

- The utility value is calculated based on the counts of different patterns of pieces on the board.

- The counts are multiplied by different weights and added or subtracted to the utility value:
  - `count(board, 4, p) * 1000`: Counts the number of occurrences of a pattern of 4 pieces in a row/column/diagonal for the current player (`p`) and multiplies by 1000.
  - `count(board, 3, p) * 100`: Counts the number of occurrences of a pattern of 3 pieces in a row/column/diagonal for the current player (`p`) and multiplies by 100.
  - `count(board, 2, p) * 10`: Counts the number of occurrences of a pattern of 2 pieces in a row/column/diagonal for the current player (`p`) and multiplies by 10.
  - Similar calculations are done for the opponent (`p2`), but with larger negative weights (-1100, -110, -10) to penalize the opponent's patterns.

Algorithm Performance

My algorithm performs differently under given time constraints. With an extended duration, the algorithm benefits from increased computational resources, facilitating deeper analysis. Ideally, a 10-second timeframe yields optimal performance, with 5 seconds considered average and 3 seconds resulting in suboptimal outcomes. Variations in performance may arise due to concurrent tasks on the computing device; hence, rigorous testing under controlled conditions, preferably on a specific platform or uniform resource, is preferable.

Human vs. Algorithm

The algorithm's limited depth hinders it from finding the optimal path, as humans can potentially be more effective with no restrictions.

Algorithm vs. Itself

When the algorithm plays itself, the following findings are observed:

- With the evaluation function defined above, the player who plays first is winning.
- With an evaluation function where it penalizes less severely when the opponent has a favorable choice, the player who plays second is winning.

Evaluation Function Comparison

With the evaluation function described at the top:

- Player who goes first wins.

With the evaluation function but with negative weights (-1000, -100, -10 in comparison to 1100, 110, 10) to penalize the opponent's patterns:

- Player who goes second wins.