Playing 3D Connect-4 Using Monte Carlo Tree Search

Willie Jeng
109550173
Artificial Intelligence Capstone
National Yang Ming Chiao Tung University

May 9, 2021

1 Introduction

The task was to make a program that plays a modified 3D version of Connect 4. Instead of the winner being the first player to reach four pieces in a row, the k-th line of 4 gives $\lfloor \frac{100}{k} \rfloor$ points to the player that made the line. The player with most points after 64 total moves (from both players) wins the game.

2 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) was used to make the AI. There were many advantages of using MCTS over other algorithms like minimax search. First, MCTS does not require any knowledge of the game besides the rules and the end condition, which means there is no need to devise a scoring system to evaluate each game state or game ending. Second, this game has a high branching factor (there are at most 24 possible moves at each turn), which would have been highly inefficient for any tree-related searches and would not be able to be evaluated for many plies.

3 Implementation

The program was written in C++, and uses object-oriented programming. The 3D Connect-4 game itself was represented using the `State` class, and MCTS was implemented with the `MonteCarlo` and `MCNode` classes.

3.1 The Game State

The game state is represented with the `State` class. It keeps track of the board state, the current score, whose turn it is, how many turns have been made, and which moves are legal.

The board state is represented as three 6×6 arrays. One array keeps track of the black pieces, one keeps track of the white pieces, and the last one is a “mask” that keeps track of all the pieces. Each cell is represented as a binary number. For example, for the white array, a 1 in the 1’s place means a white piece is at the bottom of the cell. For the black array, a 1 in the 4’s place means a black piece is at the third from bottom position of the cell. This way of representing the cells has some advantages. Besides bit manipulation being more memory-efficient and time-efficient, the height of a new piece onto a cell can easily be calculated as its mask added by 1.

Moves are represented as a number between 1 and 24 inclusive, each number representing a legal cell, which is mapped to the row-column coordinates of the cell using a constant array. The advantage of using numbers to represent moves is that it is easier to iterate through them.

Dropping a piece onto the board is done by calling `makeMove(int)`. This method adds a new piece to the corresponding cell and updates the scores, turns, and legal moves. To update the score, the current position and height is expanded in 13 directions to determine how many lines of 4 that include the newest piece are created. Using this method instead of checking for new lines aross the whole board, checking for a new score only needs $13 \times 6 = 78$ iterations in the worst case, where 4 new lines are created in every direction, which is impossible.

3.2 MCTS

The tree itself is represented with the `MonteCarlo` class, and each node on the tree is represented with the `MCNode` class.

The `MonteCarlo` class contains only the pointer to the root node of the tree. It includes the methods for completing one iteration of the search, which are `select()`, `expand()`, `simulate()`, and `backtrack()`. It also contains the method `bestMove()` that returns the best move at the game state of the root, which is the child of depth 1 that has been simulated the most times.

The `MCNode` class contains the game state of the node, a pointer to its parent node, and an array of pointers to its children. It also keeps track of the node's total reward and won reward. 2 is added to the won reward in the case of a won simulation, 1 for a tied simulation, and 0 for a lost simulation. The total reward is twice the number of simulations that have been run on the node.

3.2.1 Selection

A reference to a `MCNode` pointer is passed into the `select()` function. After running the function, the pointer will be pointed to the node in the tree that will be expanded. Starting from the root node, a node is selected if there are legal moves after its game state that have not been expanded upon. Otherwise, the same procedure is repeated using the current node's child with the highest Upper Confidence Bound 1 (UCB1) value.

The formula for the UCB1 function is:

$$\frac{w}{r} + c\sqrt{\frac{\ln r_p}{r}}$$

where w is the won reward of the node, r is the total reward of the node, r_p is the total reward of the parent node, and c is the exploration parameter, which is set to be $\sqrt{2}$ for this program.

3.2.2 Expansion

The `expand()` function uses the same node pointer reference from selection, which points to the node to be simulated on after running the function. This node is a random legal unexpanded move chosen from the selected node from the previous step.

All random numbers in this program are generated using `std::mt19937` of the C++ Standard Library, which is an implementation of the Mersenne Twister pseudorandom number generator.

3.2.3 Simulation

In the `simulate()` function, the game state of the node expanded in the previous step is simulated until the end of the game by making random legal moves. After reaching the end of the game, the outcome of the simulation is returned for backtracking.

3.2.4 Backtracking

In the `backtrack()` function, the won reward and total reward of the expanded node is updated according to the simulation results. All the ancestors of the node is also updated, up until the root node.

Each node is updated in the perspective of the opponent. That is, if the current node's game state is black's turn and the simulation resulted in a white win, the node's won reward would be incremented by 2. The reason for this is because the won reward of a node is used to choose a move for its parent, which is in its opponent's perspective.

3.2.5 MCTS using static memory

When testing the program, there were issues with memory that caused the dynamically-allocated `MCNode` objects to not be deleted properly, which caused the program to run out of memory and crash when too many iterations were performed.

To solve this, an alternate version of the `MonteCarlo` and `MCNode` were made that used a statically-allocated array to store the nodes of the tree, called `MonteCarloS` and `MCNodeS` respectively. Instead of nodes storing pointers to its children and parent, the indices of these nodes in the node array were stored instead. After testing, it was found that this method was slightly slower than the original method, but also did not have memory issues.

3.3 Main function

The template for TCP communication uses the game board state and the current turn as parameters for the solving function `getStep()`. To convert these parameters to a common game state variable, the `State` object is declared globally, and during each call of the solve function, each cell is iterated through to find the modified cell, after which `makeMove()` is run on the game state. If there are less pieces on the board than the previous state, a new game is detected, and the global state variable is reset.

Using the statically-allocated MCTS classes, it was found that a little more than 300,000 iterations could be run in under five seconds, which is the time limit for a move. However, 150,000 iterations were found to work the best and minimize the risk of memory issues and the program malfunctioning.

4 Observations

Since a close win is valued the same as a landslide win, the program often does not play optimally after a win is guaranteed or almost guaranteed. When tested against a program that makes random moves, the program usually gives up points near the end of the game after leading by more than 200 points. It would be interesting to see how differently the program would act if the goal was not just to win, but win by the most points. The definition for the won reward and total reward would have to be modified to achieve this.

Research on the disadvantages of MCTS showed that MCTS is weak at detecting a critical move that could lead to a loss. Since all of the evaluations of MCTS are probablistic, it is difficult for MCTS to know if a certain move has no way of resulting in victory. This is disadvantage is perhaps the cause for AlphaGo's loss in its fourth game against Lee Sedol. Unlike Go, however, there is not as apparent of a way to make a critical move in 3D Connect-4, and the modified scoring system for this program makes mistakes slightly easier to recover from.

5 Conclusion

With enough iterations, Monte Carlo Tree Search is highly effective at making winning moves at 3D Connect-4. The program itself was easier to design than a minimax search, as no evaulation function was needed, and the program is also more efficient than minimax search for 3D Connect-4 because of the game's high branching factor and MCTS's use of randomness to predict a move's effectiveness.

A Source code of the project

The source code of the project is in this Github repository: