COMP 424 Final Project with Monte Carlo Tree Search

author: Claire Yang (260898597), Yunya Lu(260895526)
instructor: Jackie Cheung, Bogdan Mazoure

Abstract

The project is designed to implement an intelligent student agent to play the Colosseum Survival game. Each turn, the players will go to a point on the chessboard to place a wall to make their areas as large as possible. We choose our step for each turn using the Monte Carlo Tree Search (MCTS) algorithm. According to the algorithm, we develop a Monte Carlo search tree with all nodes holding a possible game state at the beginning of the game, and use the tree so search for a step that is relatively advantageous to take. The result shows that our student agent beat most of the random opponents. The algorithm can be improved by searching for more than one node to avoid invalid steps, or by finding some prior rules to improve the efficiency and quality of the search tree.

1. Introduction

In this project, we develop a student agent that can play the game Colosseum Survival automatically with a high winning rate. The game has two agents randoms with some initial barrier walls on a square board. Agents move one by one to a reachable coordinate on the board and choose one of the four directions(up, down, left, right) to put the barrier. A coordinate is said to be reachable if the agent's path to it is free of barriers and within a specified number of steps (one step means to change its coordinate by 1). Also, the adversary cannot be in that coordinate, and we cannot set walls in the direction that already has a barrier on it. The game ends if the two agents are entirely separated by the barriers into two parts of the board. Then calculate the area of the part of each agent. The agent with a larger part of the board wins. If the two agents' parts are of equal areas, that game will be recorded as a tie.

Here we use the Monte Carlo tree search to decide the steps to take and barriers to set in the student agent. Monte Carlo tree search is a heuristic search algorithm for some decision processes [3]. It can keep our agent in the best position to win the game.

2. Motivation

Colosseum Survival is mainly turn-based moving and placing barriers until the two players are completely separated on either side of the barrier. Due to the randomness of the initial position and the opponent's action, it is difficult to find the rules of each step. Therefore we chose to generate a large number of simulated game steps to find the relative optimal solution as our rule. It happens that Monte Carlo Tree Search is suitable for such a large search space.

It is also worth mentioning that the game's mechanics are similar to Go— a game in which Monte Carlo tree search is heavily used as its artificial intelligence algorithm.[1]

3. Technical Approaches

The Monte Carlo Tree Search is mainly to build a tree with random situations and search for the most advantageous tree nodes.

For the tree node, we used the `MCNode.py`, which contains a `MCNode` class to record the state of the game after every move and barrier-setting.

To build the Tree with the node, we develop a `MCTree` class inside the `MCTree.py`. The tree-building method separates the method into four approaches: selection, expansion, simulation, and backpropagation, which you can find detailed information in Section 4. At the start of each game, we generate a root node for the initial state, which we select as the current node. For the current node, we check if it is a leaf. If it is a leaf and the game is not finished, we call expansion to generate children for the current node. For each new child, we simulated the entire game depending on its state to see the result. Then we do backpropagation to record the result on the new child node, its parent's parent, and so on, until we go back to the root. Then we select the best among all the new children to be our next current node, depending on the weights of the node, where the weight is defined in equation (1). Repeat these approaches until we cannot expand any children or the run time exceed a limit.

After completing the construction of the tree, we can start our search for the next step to take. We are still looking for the node with the highest weight in each level of depth of the tree. The weight is defined to be:

$$Q^{\oplus}(s,a) = Q(s,a) + c\sqrt{\frac{log(n(s))}{n(s,a)}}, \tag{1}$$

$n(s)$ is the number of visits;
$n(s,a)$ is the number of visits of the parent node;
$Q(s,a) = \frac{score}{n(s)}$ is the empirical weight of the node,
where $score = number\_of\_wins + 0.5 * number\_of\_ties$;
c is an constant, for example here we choose $c = \sqrt{2}$.

4. Theoretical Background

The four approaches for building the Monte Carlo Tree is:[2]

1. Selection: In the beginning, the current node is our root node. Then select the most optimal child nodes with tree policy as our next current node until a leaf node is reached.

2. Expansion: If we reach the leaf node, generate some children nodes randomly.

3. Simulation: Simulate the game after the node expanded randomly, by default policy, until we reach the end of the game and get a result.

4. Backpropagation: Use the result from the simulation to update the information of the node we expanded, and its recursive parent node.

Tree policy here is to choose our best child with weight function (1). The weight function can help maintain a proper balance between the exploration (not well-tested actions) and exploitation (the best actions identified so far).[4]

The default policy is more casual. Here we just randomly generate steps and barriers.

5. Evaluation

5.1 Advantages

- The search can be very fast after the tree is built. As in each level, only one node (highest weight) has non-empty children nodes, which is because we only select one best child before expansion. The selection with only one node also saves memory spaces.
- No prior knowledge (e.g., explicit evaluation function) is required, because we generate and simulate tree nodes freely by default policy.
- In most cases, we can find a position of relative advantage that not only has a greater chance of winning, but also takes into account the probability that the node is reachable. Since both of these are taken care of by our weight function.

5.2 Disadvantage

As the depth and number of children nodes are limited, our sample cannot cover all situations.

- As we cannot predict what our opponent will do, maybe after a few rounds, our best child in the tree is out of its optimal position.
- More seriously, our next step may no longer be valid. For example, our opponent might take our next step.
- In some extreme games, the number of turns might exceed the depth of the tree. This will deprive us of options in the part beyond the depth of the tree.
- We can only find our relative optimal step among the data we generate, which might not be the absolute best position in the whole game.

6. Improvements

As mentioned in the disadvantage part, the following steps our algorithm find is not always valid. This can be improved by returning a list of children nodes whose weight is sorted from largest to smallest. Thus if our best child is not reachable, we can still go to the next best position.

Also, we can guess some prior rules instead of generating children randomly in expansion, like taking positions in the center of the board first. This can speed up the building of the tree and improve the empirical weights ($Q(s, a)$) of the sample node, as we are closer to the more promising part in the early part of the game.

7. Conclusion

The student agent we developed with Monte Carlo Tree Search can defeat most random agents. The advantage of our algorithm is that it searches for the next step quickly and does not take up too much memory. Also, it does not need to know the game's rules to gain an advantage. However, due to the limitation of the sample size of the tree, we cannot cover all situations. Thus there are still going to be some losses from time to time. We came up with two ways to improve: one is to make the search process return more than one possible step, and the other is to add some human intelligence, like preempting the middle of the board.

References

[1] Guillaume M. J. B. Chaslot, Mark H. M. Winands, and H. Jaap van den Herik. Parallel monte-carlo tree search. In *Computers and Games*, pages 60–71, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-87608-3.

[2] Ankit Choudhary. Introduction to monte carlo tree search: The game-changing algorithm behind deepmind's alphago, January 2019. URL `www.analyticsvidhya.com/blog/2019/01/monte-carlo-tree-search-introduction-algorithm-deepmind-alphago`.

[3] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, and Sifre. Mastering the game of Go with deep neural networks and tree search, January 2016.

[4] Maciej Swiechowski, Konrad Godlewski, Bartosz Sawicki, and Jacek Mandziuk. Monte carlo tree search: A review of recent modifications and applications. *CoRR*, abs/2103.04931, 2021. URL `https://arxiv.org/abs/2103.04931`.