COMP 424 Final Project Game: *Colosseum Survival!*

Authors:

Kevin Liu: 260916466

Ruoli Wang: 260833864

Course Instructors:

Jackie Cheung and Bogdan Mazoure

1. Introduction

Colosseum Survival Game is a 2-player turn-based strategy game. The main goal of the project is to build a student agent as a player of the *Colosseum Survival Game*. In order to maximize our win rate with limited resources, we choose Monte Carlo Tree Search (MCTS) algorithm for our implementation. In addition, we add some heuristics as an auxiliary method to make MCTS more likely to produce an optimal solution. Eventually, we win nearly 100 percent of our games against the random agent.

2. Motivation

The student agent in our program is designed basing on the Monte Carlo Tree Search. Since the given time for set up is only 30 seconds, and the time to choose a subsequent move is restricted within 2 seconds, the student agent must find a solution in a relatively short time. As a result, exhaustive algorithms which are expensive may not work. As opposite to exhaustive algorithms, the Monte Carlo Tree Search using random sampling with changeable sample size will not search all states but choose the state with better prospect basing on the sampling. Thus, it can properly fit the time constraints. This is the reason why we prefer Monte Carlo Tree Search.

3. Brief Description of the Monte Carlo Tree Search

"Monte-Carlo Tree Search (MCTS), illustrated in Figure 1, is a best-first search technique which uses stochastic simulations. MCTS can be applied to any game of finite length. Its basis is the simulation of games where both the AI controlled player and its opponents play random moves, or, better, pseudo-random moves." (Chaslot et al., 2008)[1]. MCTS can infer a good strategy from a multiple simulation. The Monte Carlo Tree Search uses and maintains a tree of states according to the following four steps:

- **Selection:** This step uses a tree policy which searches a node in a way that balances between exploitation and exploration. The tree policy on one hand searches for the

node that leads to the best result (exploitation). On the other hand, due to the uncertainty of the evaluation, the tree policy must visit nodes that are less promising (exploration).

- **Expansion:** Expand the tree when a leaf node of the tree is selected.
- **Simulation:** Sampling the simulation of the rest of the game using a default policy (random move in most cases) for both players.
- **Backpropagation:** Once the simulation is done, update the value (result of the game) and visit counts for all states visited during selection and expansion.

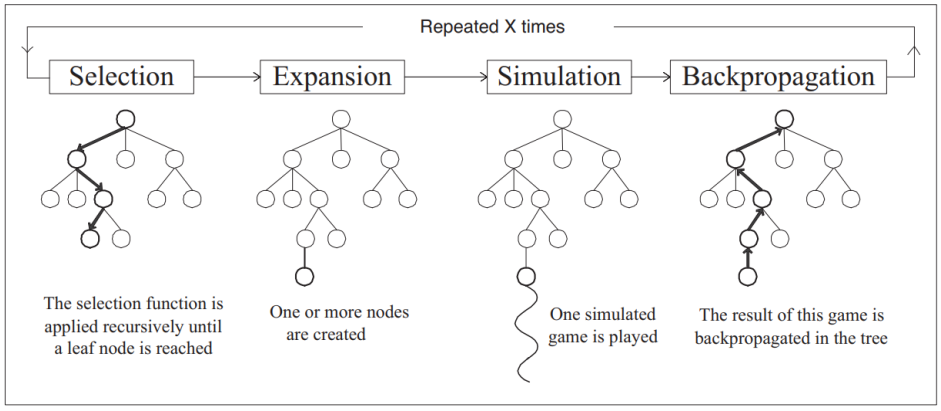

Figure 1: Outline of a Monte Carlo Tree Search (Chaslot et al., 2008) [1]

4. Technique Details

This section specifically describes how we applied Monte Carlo Search Trees inside this project.

- **Simulation:** In this part, we implements the simulation of the game by using the *World Class* in **world.py**. More specifically, we calculate the result by using the the *check_endgame* function that computes the number of blocks in each agent's zone. Due to the large searching space of a state, the sample size will be small within limited time. So, we choose to simulate three subsequent moves, which reduces the searching space significantly. After each simulation, the algorithm computes the score of the simulation by using the following equation:

$$Score = \frac{Score_{p1}}{Score_{p1}+Score_{p0}}$$

$Score_{p1}$ and $Score_{p0}$ represent the number of blocks controlled by player $p1$ and $p2$. Searching three subsequent moves makes the algorithm finish every simulation in a short time and sample more states under the time constraints. However, the game may not end in 3 moves. This does not give the algorithm comprehensive information. In this case, the score is set to be 0.5.

- **Selection:** In the simulation of candidate positions and barrier directions, the UCT is defined as follows:

$$UCT = \frac{P_{score}}{n_v} + c * \sqrt{\frac{\lg n}{n_v}}$$

In the equation, P_{score} represents the sum of score of all simulations of the node. n_v represents the number of simulations of the node. In order to balance the exploitation and exploration, an adjustable scaling constant c is introduced to give a bonus to the nodes that the algorithm has not tried too much. n is the number of times of total simulations. At the end of the game, the algorithm returns the node with highest n_v value.

- **Expansion:** For convenience, we only consider all candidate positions in the initial stage and set them as the child node of the root. And we set the UCT of each node with value $P_{score} = 1$, $n_v = 1$ at the beginning.

- **Backpropagation:** Since the algorithm only expands node of depth 1, there is nothing more to backpropagate.

5. Advantages and Disadvantages of MCTS

- **Advantages:**

1. The evaluation function of the algorithm depends only on the observed outcome of simulations. It will continually improve to the optimal (in the case of infinite memory and computation). And we can avoid constructing complex estimation functions.
2. Time and space efficiency. The search develops in a selective, best-first manner. It expands promising regions of the search space much deeply basing on the UCT function, so it runs really fast.
3. Since the algorithm focuses on the search breadth, and it only simulates 3 subsequent moves. Therefore, it can produce more accurate sampling within 3 moves.

- **Disadvantages:**

1. Due to the limit of the sample size, there exists deviations in the result of sampling. The algorithm may not produce the optimal solution.
2. The solution produced by the algorithm fully depends on the result of simulations, short of expert knowledge and experience.
3. Since the algorithm only simulates three subsequent moves, the algorithm is unable to produce a long-term strategy.

6. Future Improvement

- Increase the depth of expansion. The current expanded depth is only 1, that is, most of the simulations are performed by random sampling. After increasing the extended depth, the search tree can theoretically give a better simulation result.

- Give different weights to the nodes. If all nodes are selected with equal probability, the strategy is not ideal. We should give larger weights to nodes that looks more promising. The weights are determined basing on heuristics such as expert knowledge or knowledge learned by algorithms.
- Use Convolutional Neural Networks (CNN) and other methods can be learned by machines to replace UCT functions. The current state can be taken as the input of the CNN, and the score as the output.

7. Other Algorithm Considered

Initially, we considered the Minimax Search, which can produce the optimal solution. While there are some serious defects of Minimax Search:

- Minimax Search is an expensive algorithm, even if we apply the alpha-beta pruning. It needs to search all states of a complex game, resulting in a large branching factor.
- Minimax Search assumes that both players take the same evaluation function.
- Minimax Search needs a good evaluation function. However, it is hard to design an evaluation function especially when there are only a few moves.

References

[1] Guillaume Chaslot, Sander Bakkes, Istvan Szita, and Pieter Spronck. Monte-carlo tree search: A new framework for game ai. *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 4(1):216–217, Sep. 2021. URL `https://ojs.aaai.org/index.php/AIIDE/article/view/18700`.