COMP 424 Final Project Game: *Colosseum Survival!* Report

Author: Yimei Yang (260898303) and Céline Wu (260905349)
Due Date: April 12th, 2022, 11:59PM EST

1. Technical Approach

The program is designed to output the best step when competing with an opponent player. To do so, we choose to implement the Monte-Carlo Tree Search to help us decide which position and direction would have an optimal result. Specifically, there are four main phases – selection, expansion, simulation, and backpropagation.

1.1 Basic Structure

1.1.1 Selection

Use a tree policy, the Upper Confidence Tree, to balance exploitation and exploration when selecting a promising path in the search tree. We select the best node and repeat the process, "moving down the tree to a leaf" (Russell and Norvig, 2010).

1.1.2 Expansion

Expand the tree to get all legal actions when we are at the frontier where the current state has no children.

1.1.3 Simulation

Use a default policy to sample rollouts randomly by alternating player turns. These plays are not recorded and updated on the search tree.

1.1.4 Backpropagation

After a simulation is completed, "update value and visit counts for states visited during selection and expansion". (Jackie CK Cheung and Bogdan Mazoure, Monte Carlo Tree Search, Lecture 9 slides)

1.2 Main step function logic

In the step function of the student agent where we calculate which step to take, we do the following to combine the four stages recursively. For a given node, we first examine if the node has children. If it does not have children, we select the best node based on their UCT score. If the node has children, we further examine whether this node has been simulated or not. If not, simulate this node and backpropagate to get a value showing how promising this node could win. If the node has been simulated before, expand it by finding all its legal actions. Finally, we will get the best node by returning to the root node and find the best actions by iterating through the UCT value of its children. During the whole process,

we have implemented a variable to keep track of whose turn it is to know whether it is the adversary or myself to make the first move in the expansion. The program will terminate if time exceeds 1 second or we have reached two levels of the tree using the tree policy.
We have referenced the starter code in the world file and simulator file when implementing the algorithm. Specifically, in the expansion phase where we try to find all legal actions, we have referenced the check valid step function. Moreover, in the simulation phase where to utilize the default policy, we have referenced the random walk function to simulate random steps. The whole simulation process also takes reference from the simulator.

To ensure the step function can terminate within 2 seconds, we imported the time module using the time() function. The logic we employed in the step function is a while loop. The time function becomes a helpful tool to terminate the while loop and ensure the efficacy of this function at the same time. The function time() records the exact time when this line executes. For example, the start_time is 7:30:20. Each time() executed in the while loop must be later than 7:30:20. If the time() reaches 7:30:21, the loop will terminate.Therefore, by examining the execution time in each loop, we can terminate this loop as soon as the execution time is larger than 1 second.

```
start_time = time()
while True:
    if time() - start_time >= 1 or d > 3:
```

Figure 1: Time() in step()

We used the logic in the simulation step. We want to maximize the number of simulations while terminating the simulation function within a reasonable time. By trial and error, we found that 0.5 seconds is a safe ground where board sizes under 7 can finish 5 simulations and board sizes larger than 7 also have the chance of getting simulations.

```
startTime = time()
while (i != 5):  # check time,
    if time() - startTime >= 0.5:
        break
```

Figure 2: Time() in simulation()

2. Motivation for Technical Approach

There are many approaches to solve this problem, including brute force, MiniMax, MiniMax with alpha-beta pruning, and Monte-Carlo tree search. Before trying to use the Monte Carlo Tree Search as our game algorithm, we attempted to implement Minimax search and alpha beta search. However, we decide not to use any of them with various reasons.

2.1 Brute Force using BFS and DFS

The problem with brute force is that the game space is too large to iterate through every possible state. Especially with the 12x12 chessboard, the branching factors can be up to 312. Considering the time limit and space limit, using this method is impossible.

2.2 Problem with the Minimax

we found that it is difficult to figure out a good evaluation function. Also, it is necessary to iterate through all the branching factors of a given action. Also, we need to find an appropriate evaluation function. Even if we implement the Minmax search, our evaluation function is inaccurate, the result is still not optimal. We could use MiniMax cutoff to ensure the time and space limit. However, a good evaluation function is still required.

2.3 Problem with alpha-beta pruning

By using the alpha-beta pruning, we could prune many non-promising branches and save a lot of time. However, the problem of evaluation function is still not solved. Similar to the Minimax,it is till impossible for us to develop an evaluation function. Moreover, the Colosseum Survival game would have a large number of branching factors, which alpha beta search will be limited (Russell and Norvig, 2010).

2.4 Why Monte Carlo Tree Search

Firstly, Monte Carlo Tree Search does not require an evaluation function as it only relies on simulation results and can still find a promising move. We could use UCT which automatically evaluates how promising it is for each state(node), which is more optimal than than the one we manually coming up with by ourselves. Secondly, because of the UCT, we can balance exploration and exploitation. It allows us to continue to evaluate other alternatives by executing them instead of the current perceived optimal strategy. Especially when we facing a large number of simulations, UCT would still be very effective.

3. Pros/cons of Chosen Approach

3.1 Pros

- Monte Carlo Three Search converges to the Minmax solution. (Jackie CK Cheung and Bogdan Mazoure, Monte Carlo Tree Search, Lecture 9 slides)

- It balances the exploration and exploitation trade-off, avoiding the issue where the current best action is not the overall optimal action.

- We can interfere with what extent the algorithm goes to satisfy the time limit.
- The time to compute a playout is linear, not exponential, in the depth of the game tree, because only one move is taken at each choice point. That gives us plenty of time for multiple playouts. (Russell and Norvig, 2010)
- the reinforcement learning nature allows us having no previous knowledge or strategy about the game but can still develop a good policy to win the game.

3.2 Cons

- Using the tree policy could be time and space-consuming if we do not interfere with when to stop.
- It requires a large number of iterations to decide the most efficient path.
- when the chessboard size goes big, i.e. 12, the branching factor became really big. It would take too long to select the best move.

4. Future Improvement

There are mainly three parts we can improve.
The first one is the while termination. For some reason, the time() function does not ensure that the step function can finish within 2 seconds. Therefore, many random walks are going on in the auto-play mode. We tried to shorten the time limit for the while function. However, we saw no noticeable improvement.
The second part is the simulation function. For some reason, simulations are stuck into an infinite loop in some cases. Even though we implemented the time limit inside the simulation function, we observed that the simulation function does not stop as we intended in some cases. We tried to debug the simulation function and figured out some extreme cases in which the agent is stuck in a place where all four directions have walls. Because the agent always tries to find a valid step to take, the agent is destined to be in an infinite loop. However, we did not have enough time to explore how to solve this problem and used a time limit to break the infinite loop in force instead.
The third part is that we did not have the chance to explore the benefit of alpha-beta pruning. We did not compare the efficiency of MiniMax+alpha-beta pruning+cutoff with the Monte-Carlo tree search cutoff. If we have more time, we can utilize the idea of alpha-beta pruning to decrease the number of actions we expand so that a large board size(larger than 7) has a lesser chance of taking the random walk.