COMP 424 Final Project Report

Course Instructor: Jackie Cheung and Bogdan Mazoure
Project TAs: Koustuv Sinha (koustuv.sinha@mail.mcgill.ca) and Shuhao Zheng (shuhao.zheng@mail.mcgill.ca)
Due Date: April 8th, 2022, 11:59PM EST

Authors: Ada Andrei - ID: 260866279, Christian Martel - ID: 260867191

1. Implementation

1.1 Algorithm

1.1.1 MCTS

Our agent uses a Monte-Carlo Tree search algorithm in order to find its next move. Our MCTS implementation consists of a tree where the root node corresponds to the game state before the agent's move. Each node has a visit count and a win score used for the promising node choice. Also, each node has a game state. A state includes the board barriers locations, the positions of each player and the player turn. The children of a node consists of all possible valid nodes reachable from the node's state.

1.1.2 Node Expansion

A node is expanded by generating a child node for each possible valid move from the state of the node. These moves are generated using a breadth first search algorithm. Each node that is reachable within the maximum number of steps without walking over a barrier or without walking over the opponent is added to the list.

1.1.3 Simulations

The MCTS tree is initialized during the first round. The agent is assumed to be *P0*. The algorithm performs as many simulations as it can in the provided time for the round. Each simulation starts with the selection of a promising leaf node from the root node of the MCTS tree. The node selection follows the principle of upper confidence trees further explained in section 2. The promising node is expanded and a random child is selected from the generated children. Then, a random play-out is performed on the chosen child until a terminal state is reached. There are four terminal state results possible: *P0_WIN, P1_WIN, DRAW* and *IN_PROGRESS*. The result is back-propagated until the root node.

1.1.4 Back-propagation

During backpropagation, for each node visited in the path to the root, the visit count and the win score are updated according to the play-out result. Here are the scores added to the win_score depending on the play-out result:

Status Code	Added Score
P0_WIN	1.5
P1_WIN	0
DRAW	0.5

Table 1: Score added for terminal state statuses.

1.1.5 Root Update

The root of the MCTS tree is updated at the beginning of each subsequent turn. From the root node, a breadth first search algorithm is applied to find the node that corresponds to the new current state. If state is found, it is set as the new root node. Otherwise, a new node is created and set to be the root.

1.1.6 Trap Detection

To make our simulations more realistic, a trap detection mechanism was added to filter out bad moves in node expansion and random playing. A trap is a move where the agent ends up being surrounded by 3 or more barriers. We want the agent and the random plays to avoid these kind of moves.

1.2 Results

1.2.1 Performance Against Random Agent

After running 1000 random games against the random agent, our algorithm achieved a 97% winning rate. The average game time of the 3% of losses is 3 ms. We suspect that theses losses are due to a random initial game setting that is terminal in which our agent loses.

1.3 Motivation

The MCTS search was chosen since the Coliseum Survival game is a game with a very high branching factor and many moves per player. Assuming a board without any barrier, the branching factor complexity is $b = \mathcal{O}(4 \times 4 \times (3)^{K-1})$ where K is the maximum number of steps. The number of moves per player complexity is $d = \mathcal{O}(2 \times 2 \times (B - 1) * B$. For example, on a 8x8 board, with K=3, there would be an upper bound of $b^d = 5.4525431940768203990886802379611e + 241$ possible play-outs.

Optimal algorithms like minimax or alpha-beta pruning would definitely be to expensive and would not have time to converge in the simulation time allowed. Hence, we chose to implement a Monte-Carlo Search tree that would allow us to find a good estimate of the optimal next move.

1.4 Hyper-Parameter Tuning

Some hyper-parameter tuning was achieved by running play-outs of our agent against himself with different parameter values. We tuned the win and draw scores using this strategy.

2. Monte-Carlo Tree Search

The Monte-Carlo Tree Search Algorithm is a search algorithm that expands nodes based on simulations of the game. It starts with a simple search algorithm and has the algorithm play a certain amount of games against itself using a specific policy to select actions deterministically or stochastiscally. Its evaluation function also depends only on the observed outcome of the simulations and it continues to improve from additional simulations. There are four main parts of this approach as described in the course notes:

1. Selection: Use a tree policy on nodes seen before to select a promising path in the search tree.

2. Expansion: Expand the tree when we reach the frontier.

3. Simulation: From an expansion node, sample playouts. (rollouts) using a default policy for both players.

4. Backpropagation: After the rollout reaches a terminal node, update value and visit counts for states visited during selection and expansion.

To select the next move in the search tree, we base our search on Upper Confidence Trees (UCT) which compute a value for taking an action for a given state. This is how we define our tree policy.

2.1 UCT

$$Q^{+}(s,a) = Q(s,a) + C \times \sqrt{\ln \frac{n(s)}{n(s,a)}} \tag{1}$$

$$Q(s,a) = \frac{w(s,a)}{n(s,a)} \tag{2}$$

$$C = \sqrt{2} \tag{3}$$

The first term in equation (1), $Q(s,a)$, is the exploitation term. It is the UCT value of taking action s from state s. It is computed by dividing the win score of taking this action ($w(s,a)$ by the total number of times the action was chosen through simulation ($n(s,a)$). The second term in equation (1), $C \times \sqrt{\ln \frac{n(s)}{n(s,a)}}$, is the exploration term. It gives bonus to actions that were not chosen a lot.

3. Pros & Cons

There are multiples advantages to using a Monte-Carlo Tree search algorithm which is why we chose to implement it for our agent.

One of the advantages of Monte Carlo simulations is that as we approach an infinite number

of simulations, the MCTS converges to the optimal search tree solution.

The solutions to the Monte-Carlo simulations also improve with the lines of play which we can control to match our allotted time to pick a move. Finally, the different simulations could easily be parallelized, but we did not parallelize our program. We can expect that our agents performance would improve with parallelization because we would be able to increase the number of simulations we run.

However, there are a few disadvantages to our approach. The Monte-Carlo approach we employed can miss optimal plays when they are at deeper nodes that we do not search since we have a time limit. In addition, using random plays as opposed to is likely not the most effective policy for selecting candidate plays, but our trap detection mechanism does filter out some bad plays.

4. Other Approaches

The first alternative approach we tried was adding Minimax plays when there were few possible moves left. The Minimax algorithm computes the minimax decision from the current state, and minimax values of each successor state all the way down to the leaves of the tree such that each play maximizes the minimax value of the player throughout the entire the game tree. However, it did not converge since there were too many possible scenarios to investigate. Since we were already constrained by the time limit, this alternative did not end being suitable.

The second alternative we employed was improving the random selection policy by avoiding moves that would lead to immediate loss, but this increased the time per simulation which in turn allowed us to realize less simulations and get a worse performance.

5. Improvements

There are a few improvements that could be included in our algorithm. Firstly, our approach could be improved by performing parallel simulations. Performing more simulations in the same amount of time would make our move estimate converge towards the optimal move. This would allow us to add filters as needed to prioritize good moves and avoid bad ones. An example of a weighted filter could give more weight to moves that are closer to the center versus those on the sides.

Additionally, heuristics could be used to evaluate move selection. For instance, Heuristics could be a function of the amount of barriers added in a player's zone.