COMP 424 Final Project Report

Hongyi Qi(hongyi.qi@mail.mcgill.ca)

Tommy Zhou(shanyue.zhou@mail.mcgill.ca)

1. Introduction

The goal of this project is to develop an AI agent for a game called *Colosseum Survival!*. Each player can take turns to move with specific steps in an $M \times M$ chessboard, then place a barrier around it at the end of the move. The barriers will eventually separate the chessboard into two parts, and the player in the part with more blocks wins.

We developed an AI agent based on Monte Carlo Tree Search(MCTS) in this project. The agent iterates to simulate the game within the 2-second decision time and returns the estimated best move based on the generated Monte Carlo tree. We carefully tuned the exploration parameter of the tree and obtained the optimum value for this board game. In practice, our agent beat the random agent by a win rate of 98% in a match that consists of 50 games.

2. Explanation and motivation

2.1 motivation

Our motivation began with formulating the process of this game into a search tree. The root node is the initial chessboard, and the nodes on each level of the search tree are all possible moves that the current player can take. The leaf nodes are terminal conditions where the game ends and outputs the winner. This search tree has a large branching factor. Given the chessboard size as $M \times M$, the max branching factor for the corresponding search tree, ignoring boundaries and the adversary position, is bounded by

$$2M(M+2)+4$$

Note that when $M = 12$, the max branching factor already hits 340. In practice, the player movements are bounded by boundaries, barriers and the adversary position, so the actual branching factor is lower than the estimated upper bound, but it is still impossible to perform a full traverse down the search tree within the time limit.

The high branching search tree and similar terminate conditions remind us of the game Go. Both games are finite two-person zero-sum sequential games with a large search tree. A well known AI-agent for Go is Alpha Go, an unprecedented AI agent that defeated human Go masters. One of the core components of Alpha Go is the Monte Carlo tree search (MCTS), which is used for the agent to choose its next move even if the branching factor is large. [2] The success of AlphaGo suggests the feasibility of implementing our agent based on MCTS, which is the primary motivation for us choosing MCTS in this project.

2.2 Implementation

We aimed to find the most promising next move given the current game state based on MCTS, which is the player's new position on the chessboard and the direction of the barrier the player puts. These parameters is the return value of the `step()` method.

Due to security reasons, we do not have direct access to the World class, which records player movements and the current chessboard. We built the class `MyWorld` that reconstructs a modifiable chessboard. The class `MonteCarloTreeSearchNode` then uses this chessboard to perform simulations and generate the Monte Carlo tree. The structure and key methods of these classes are shown in fig. 1.

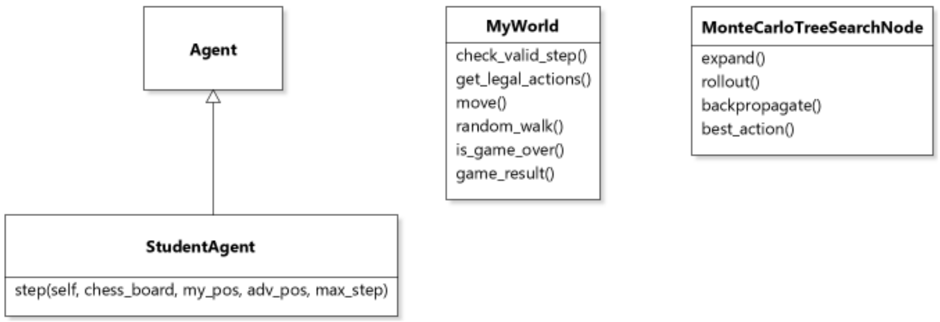

Figure 1: Class Diagram for the agent

The core function of the MCTS is `best_action()`, the pseudocode is provided in algorithm 1

```
Data: Time limit $t$
begin
    endTime = currentTime + t
    while $currentTime < endTime$ do
        $v \leftarrow$ tree_policy()
        reward $\leftarrow$ v.rollout()
        v.backpropagate(reward)
    end
    return $self \rightarrow best\_child$
end
```

Algorithm 1: Best action selection algorithm

In the `best_action` method, the algorithm iterates through the three methods to explore the Monte Carlo tree.

- The `tree_policy()` is called to select and expand the current MCTS. It will list the set of untried legal moves, use the upper confidence tree to select and create a new node v in the Monte Carlo tree, and return the new node.
- The `rollout()` will execute the `random_walk()` method to obtain a random move for each step, the `move()`method in `MyWorld` class is called to update the chessboard and iterates until the terminal condition is reached. The reward is 1 if our agent wins, -1 if losses and 0 when draw occurs.
- The `backpropagate(reward)` update the number of visits and the result for this node. It will then recursively call this function on its parent node until the function reaches the root node.

3. Theoretical basis

Many decision-making problems can be formulated into a search tree. However, the heuristic tree search has two significant drawbacks: Firstly, it cannot solve problems with high branching factors as the search space is exponentially bounded by $\mathcal{O}(b^m)$ where b is the branching factor, and m is maximum depth. Secondly, it is hard to apply a heuristic alpha-beta search if the evaluation function for the current decision is hard to define. [1]

The Monte Carlo tree search is introduced to overcome the two challenges. Instead of using a heuristic evaluation function, it estimates the average utility over several simulations that continue the game based on the default policy until the terminal condition is reached. More formally, the MCTS consists of the following four steps:

- Selection: Use a tree policy to select a promising node near the root waiting for exploration.
- Expansion: Add a leaf node based on selection to expand the tree and record future simulation results.
- Simulation: Run the default policy until the game reaches a terminal condition. This process is called rollout.
- Back-Propagation: Update all the utility values for all nodes accessed from the leaf node to the root.

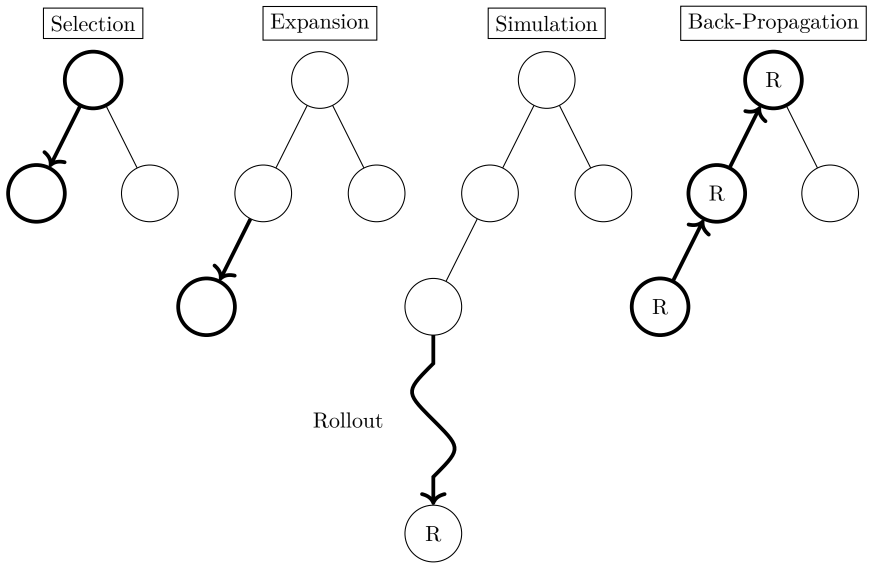

Figure 2: Process for MCTS

The visualization of the four stages is shown in fig. 2. The search algorithm iterates through the four steps before exceeding the time limit. It will then select the most promising node based on the tree to decide the next move.

The default policy for this agent is random moves, and the tree policy is the upper confidence tree (UCT), defined by:[1]

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

where $Q(S,a)$ is the value of action a in state s, $n(s,a)$ is the number of times we have taken action a in state s, c is the scaling constant that controls the chance of exploration.

4. Evaluation

Advantages

First, the implemented agent achieved a relatively satisfying result against the random agent. In a match that consists of 50 games, our agent has a win rate of 98%. The algorithm will eventually converge to the Minmax solution if given enough time.

```
   100%|--------| 50/50 [11:15<00:00, 13.56s/it]
INFO:Player A win percentage: 0.98 (13.5501 seconds/game)
INFO:Player B win percentage: 0.02, (0.0006 seconds/game)
```

Secondly, our agent is highly adaptive in various ways. One can increase the algorithm's performance by simply increasing the number of iterations of the MCTS. The number of iterations can be increased by giving more time for the agent to iterate or using processors with higher core speed. Also, if the game rules are subject to change, we only need to modify the `MyWorld` class to make sure the game simulation runs correctly. The only hyperparameter that requires tuning for the MCTS is the scaling constant.

Thirdly, the agent is unaffected by the branching factor, as we can control the number of lines of play to match the limited time. The structure of the agent is reliable as long as the branching factor is kept at a reasonable size (e.g., polynomial size). [1]

Disadvantages

One of the most significant drawbacks is that the agent needs information about all valid steps during the expansion process before choosing a new leaf node. In other games that the

MCTS approach can dominate, the set S_c, which contains all possible moves under current game configuration c, is usually very easy to obtain. For example, in the Go game, the set S_c is all coordinates that do not have a stone. However, in *Colosseum Survival*, every move requires an additional BFS search to check if the agent can bypass the barrier and reach the coordinate. This step is extremely time-consuming and takes up valuable MCTS time. In table 1, the profiler reveals that the program spent over half of the runtime checking if the node is valid instead of simulating for results.

Name	Call Count	Time(ms)
check_valid_step	443788	101089
<method 'reduce' of 'numpy.ufunc' objects>	12999940	19501
array_equal	12999599	50864
...	...	...

Table 1: Profiler of the Simulation over 10 games

A more general disadvantage for the MCTS algorithm is that it can miss the optimum play as the moves in the deeper nodes are not fully recorded.

5. Discussions and Future Improvements

The decision made by MCTS depends on the simulation result, and the results are greatly dependent on the default policy. Our agent's current default policy function is a uniform random move. However, the rollout in AlphaGo is performed by a neural network value estimation. The current node is directly evaluated with a 19-layer convolutional neural network (CNN). [2] We may try a similar approach by replacing the random default policy with a pre-trained small CNN and increasing the quality for each rollout.

Another way to improve the performance of our agent is to decrease the number of calls of `check_valid_step()` as it takes much time. One possible solution is to ignore the validity of the nodes during the MCTS. We then check the validity of the most promising node during the output phase. If the move is invalid, we move to the next most promising node until a valid node is found.

References

[1] Stuart J. Russell. *5.4 Monte Carlo Tree Search*. PEARSON, 2020.

[2] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.