COMP 424 Final Project Report

Cecilia Jiang ruixi.jiang@mail.mcgill.ca

Zhiwei Li zhiwei.li2@mail.mcgill.ca

1. Introduction

1.1 Overview

Searching, one of the biggest topics in AI, is the process of navigating from a start state to a goal state through multiple intermediate states. Search algorithms are widely used in games to figure out a strategy. In this project, our goal is to build an AI agent to play the two-player board game called *Colosseum Survival*. The choice of our group is Monte Carlo Tree Search.

1.2 Organization of Report

In this report, we will first discuss the characteristics of *Colosseum Survival* and our motivation for using Monte Carlo Tree Search. Next, we share the implementation details of the AI agent. We will then evaluate and analyze the pros and cons of our approach. Finally, the possible future improvements will be discussed.

2. Motivation for Technical Approach

In order to choose an appropriate search algorithm to develop the AI agent, we first determine the following characteristics of Colosseum Survival.

- **The game is two player turn-based.** Two players take turn to put barriers on the board until they are separated by the barriers.
- **The initial state of the game is stochastic.** The size of the board of each round varies between 6 and 12. Players A and B and initial barriers are randomly positioned on the board symmetrically.
- **The game is adversarial.** The goal is to maximize the agent's score; specifically, maximizing the number of blocks in the agent's zone in the endgame.
- **The game provides perfect information.** The current game state is visible to both players and no information is hidden.

By the characteristics of the game, Monte Carlo Tree Search and Minimax are two candidate algorithms to use to develop the agent. Monte Carlo Tree Search is a tree search algorithm that evaluates each state by simulating the game. Minimax search algorithm uses an evaluation function to evaluate all possible moves from each state. After comparing the two search approaches, the motivation for choosing MCTS is below:

- **The state space is large and the branching factor is large.** Using Minimax search results in a relatively larger tree. MCTS samples states and simulates the results instead of using a static heuristic evaluation function. MCTS spends time expanding more meaningful nodes, unlike Minimax treats all the nodes equally and thus does not waste time expanding all possible nodes.

- **Defining an appropriate heuristic evaluation function is hard.** For a state during the game, it is difficult to determine which player has a better winning percentage and which move gives superiority. Instead of relying on the evaluation function, MCTS learns rewards from random simulations without heuristic domain knowledge of the states.

- **MCTS can stop at any given time and return a decision.** As specified in the instruction, the agent is only allowed 2 seconds to choose a subsequent move. MCTS can use out the 2 second computation time and return the current best estimate by simulation results.

3. Technical Approach and Implementation

3.1 Monte Carlo Tree Search

The Monte Carlo tree search algorithm is a tree-structured approach that uses a heuristic search to provide optimal decisions based on cumulative reward feedback. The root node of MCTS represents the current state of the game and all the children nodes represent the states that might be visited in the next few steps. The mathematical core of MCTS is based on UCT, which applies multi-arm bandit ideas to guide the selection of nodes. The foundation of search iterations includes four phases which are selection, expansion, simulation, and backpropagation. Figure 1 illustrates the four phases of the Monte Carlo tree search.

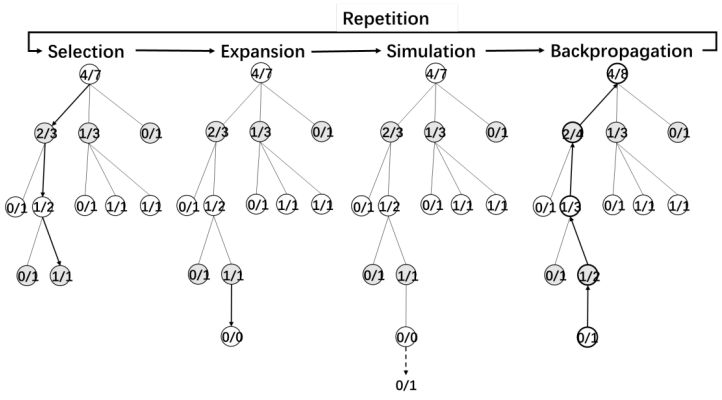

Figure 1: Monte Carlo Tree Search Flow Chart[1]

The selection phase selects the node that has maximum UCT until reaches the leaf node and selects it as the beginning node. The expansion phase selects the child node of

the selected node, and the simulation then starts. When the simulation reaches a terminal state, the winning result is backpropagated to the root node.

3.2 Our MCTS-Based Agent

An MCTS-based algorithm is used to implement the AI agent for the game. For the first step of the move, the Monte Carlo tree is created and expands for thirty seconds. After the opponent takes the step, the agent first updates the root of the tree according to the opponent's move calculated from the new board state. Then the agent develops the tree for 2 seconds and returns the best move.

Every node of the Monte Carlo tree records a certain state of the game. In our algorithm, each tree node records the move that reaches this node, number of visits, rewards, children nodes, parent node, and turn. To reduce memory usage, tree nodes record the move instead of the whole board state. The board recording has high space consumption and is not necessary since the board state can be recovered by the recorded moves. We only need to record the board state of the root node and the rest of the moves. The number of visits and rewards are used to calculate the UCT of nodes. Also, the turn records the current player of the state.

- **Selection:** The game starts with the selection phase. If the node's available moves have all been explored at least once before, the tree will set the current node to be the child node that has the maximum UCT. The UCT formula is shown below

$$UCT = \frac{w_i}{n_i} + c * \sqrt{\frac{\ln t}{n_i}} \tag{1}$$

  . The first component of the formula is the exploitation component which shows the winning conditions of the node. The second component, which is the exploration component, allows us to explore relatively rarely visited nodes so that the algorithm does not greedily explore only the nodes that bring a single winning playout. We choose C=0.5 in the selection process to encourage the agent to explore more in two seconds. If we reach a node where not all moves have been explored, the search goes to expansion phase.

- **Expansion:** The Expand algorithm is different from the standard MCTS, which tries to append all possible children nodes to the current node. Instead, the number of children nodes added is determined by its depth, which will be discussed in more detail in section 3.4.

- **Simulation:** Simulation then starts at the selected current node. Random sequence of moves will be taken until the game terminates. In order to improve the performance of the search, we rewrite the step function given in random_agent.py which will be further discussed in section 3.3.

- **Backpropagation:** When the game stimulation reaches the end, the backpropagation function will be called to update the reward and recover the board back to the root board. The reward is 1 for a win, 0.5 for a tie, and 0 for a loss. After the backpropagation is finished, the search process goes back to the selection phase to start

a new iteration. When the time is expired, the node with the most number of visits will be selected.

3.3 Key Functions

The *get_next_state* function to find possible moves for a position is implemented using breadth-first search. First, put the initial position of the node into a queue and then dequeue this position and enqueue all possible positions in one step that can be reached. To avoid stepping into the positions that have been explored, a 2D array called step record is used to record positions that have been taken. After *max_step* times of iterations, all possible positions that can be reached are recorded in the step record. The possible move now is to fill boards for these positions.

The *step* function given in random_agent.py considers the moves going back and forth. For example, the random step is three, and the agent is currently at location a. The generated random move is $a > b > c > b$. However, this result of this move($a > b$) is also a random move when random step is one. In order to improve the efficiency of our agent, we rewrite the step function in student_agent.py so the duplicated moves are not in consideration. The original step function is biased towards the steps that is close to the current location since the location near the agent's current location may be considered more than once. The modified step function gives the agent more chance to explore the larger steps.

3.4 Improvements

To improve the performance, we design a smarter opponent called *better_random_agent* which our agent can play with. The *better_random_agent* can choose the step that can win the game directly and avoid steps that might cause the opponent to win directly. We found out that during the start of the game, the number of possible moves is large and thus the simulation time for one node is not enough to detect the danger. Given that the time for the stimulation is limited and this issue cannot be easily addressed especially for large board sizes. we decide to add one calculation layer before the development of the Monte Carlo tree. This layer calculates steps that can win the game directly. The cost of this calculation is small, less than 0.1s according to experiment results, but the algorithm is very efficient for two agents using the Monte Carlo tree to play against each other. The reason to discard the calculation to avoid the possible moves that lead to one-step loss is that the execution time might exceed the required 2s for large boards, and we want to leave enough time to develop the tree.

3.5 Discussion

To have more flexibility in end game evaluation, we introduce a different approach to get game results compared with the one written in the Simulator. The new algorithm is the *get_next_move* function with an unlimited number of steps to record all the areas it can reach. If one agent meets the opponent's positions, it means that two players are still in the same region and the game is not ended. If the opponent's position is not reachable, the

algorithm will count the areas by using the step record array occupied by two players and return their scores.

During the development of the MCTS algorithm, we first append all the possible children during the gameplay process. This strategy leads to O(nm)(n is the average children node number and m is the iteration time) space consumption. This will cause the excess required operation memory size and the garbage collection process will also slow down the entire stimulation. Thus, to improve the efficiency of space usage, we introduce an algorithm that can help to reduce the node created by one game stimulation to O(m).

As the tree gets deeper, the node being created is used less frequently given the nature of the MCTS algorithm, so instead of appending all the child nodes, we use a shrink factor to reduce the number of nodes added to the tree. For nodes at depth n, $1/2^n$ of all children nodes plus one will be appended. The MCTS algorithm will first choose the unexplored nodes so the reduction of possible children will not affect the node choices. When the number of children nodes is smaller than the visiting time, an expand function will be called to add more unexplored children nodes. With the help of this algorithm, the space for single game play is reduced to m+m/2+m/4+...m/2^n, reduced to O(m). Then we found out that finding all possible moves and slicing them is much more expensive in terms of time consumption than randomly choosing a possible move of a node, so after the depth of 4, we employed the *get_one_child* method to randomly select a move to continue the game.

4. Pros and Cons of MCTS

4.1 Advantages

One advantage of our MCTS algorithm is that it is guaranteed to find a step given a certain amount of time. If the algorithm's searching time varies from time to time and cannot stop at a time, the algorithm may not finish searching in 2 seconds, therefore, returns a random step. The other advantage is that we do not need to define an evaluation for the game. The algorithm can be applied by knowing the information of the states, moves, rewards, and number of visits. Lastly, our MCTS algorithm balances the exploration-exploitation trade-off by using the UCT formula. The algorithm periodically visits relatively unexplored states and discovers potentially more optimal moves than the current visiting state.

4.2 Disadvantages

The major disadvantage of our MCTS approach is that the CPU time is related to the returned answer quality. Running more simulations gives more accurate estimation results. For the larger size of the board, the game has a high branching factor so the children nodes may not be sufficiently visited. Also, there may exist a case where the tree has extreme depth, which slows down the simulations for the move. The other disadvantage is that our MCTS approach may not always find the 'trap states' where a single branch leads to a single loss[2]. Since such cases may not be easily found by random search, the MCTS algorithm may lose. For example, if one move is good for 90% of opponent's moves but lost directly to 10% of opponent's moves. Since our strategy is random play, the danger of losing directly is not obvious to the agent and might lead to bad step choices.

5. Future Improvements

5.1 Machine Learning

Our MCTS algorithm can be further improved by machine learning approaches. Machine learning can be applied prior to the tree expanding in order to reduce the branching factor. To train the model, we need to obtain a dataset of the board states with the label of win conditions. As each state is represented by a three-dimensional array, convolutional neural networks(CNN) is a useful machine learning approach to the 2D+depth structure of data. After the model is saved, we could use the model to predict the win conditions of any given state so we could eliminate low winning percentage states.

5.2 Opponent Modelling

The standard MCTS tries to simulate the game without any consideration of opponent's actions[1]. This causes the agent to run simulations on moves that are not necessarily required. If the opponent is rational, we could apply adaptive play opponent modelling techniques to further improve the performance and efficiency of our agent. This adaptive approach learns the characteristics of the opponent's moves over time and make better decisions when exploring the tree.

References

[1] Yanxia Guan, Han Long, and Shengde Jia. Research on monte carlo tree search method of adaptive resource scheduling for multi-agent game. Changsha, China, 2021. URL http://dx.doi.org/10.1145/3474198.3478175.

[2] Sabarinath Mohandas and M. Abdul Nizar. A.i for games with high branching factor. pages 372 – 376, Thiruvananthapuram, India, 2018. URL http://dx.doi.org/10.1109/CETIC4.2018.8531047.