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

Jerry Hou Liu (260920691) and Justin Sun (260707684)

December 6, 2023

1 Motivation

Our agent for the *Colosseum Survival* game applies Monte Carlo Tree Search (MCTS) to intelligently find moves. Because of MCTS's ability to provide insights into the long-term impacts of a move without exhaustively searching through all possible moves, our agent provides fairly high performance in games, beating the random agent nearly 100 percent of the time. The game's extensive branching factor, evident in the numerous potential moves on a 12x12 board, combined with our constraints of a 2-second time limit and a 500 MB RAM cap, prompted us to choose Monte Carlo Tree Search. Although the game is deterministic and fully observable, making it suitable for approaches like minimax with alpha-beta pruning, using minimax would demand a cutoff test or an evaluation function given our constraints and the branching factor of the game. In contrast, Monte Carlo Tree Search simulates full games randomly from possible next moves and backpropagates the result, so we can pick the best move based on the win rate from the simulations. This allows more focus on subtrees that have more promising moves, eliminating the problem of the game's high branching factor. (L8, Slides 7-11)

2 Detailed Explanation

2.1 Overview of Monte Carlo tree search

Monte Carlo tree search (MCTS) is a method for finding the best move in a game that combines a *tree policy* that selects promising nodes to be

expanded with a *default policy* that simulates games starting from a node to help estimate the utility of a move. Each time a simulation is completed, the value of the node is updated based on whether the agent won or lost the game, and the updated value is back-propagated to the parent nodes. Each node tracks the number of games that begin from itself which have been won, lost, or tied, whether simulated directly from the node itself or from a child node expanded from it.

The tree policy combines the *exploration* of nodes with little information about them and the *exploitation* of nodes that data from simulations indicate to be of high utility. The following equation shown in L8, slides 44 through 48, is used in the Upper Confidence Trees (UCT) method to balance exploration and exploitation:

$$Q^{\oplus}(s,a) = Q(s,a) + c\sqrt{\frac{\log n(s)}{n(s,a)}} \tag{1}$$

Equation (1) is used to determine the next node to traverse to given the current node s. $Q(s,a)$ is the currently estimated value of taking an action a given the state s; $n(s,a)$ is the number of times we have taken the action a from state s, and $n(s)$ is the number of times we have visited state s, regardless of the next action taken. The first term represents the value of exploitation—it captures the currently estimated utility of selecting node a. The second term represents the value of exploration, and becomes higher when the number of times a has been visited from s is small compared to the number of times s has been visited. c is used as a scaling factor: a higher value increases the importance of exploration. The child node a with the highest estimated utility after adjustment for exploration $Q^{\oplus}(s,a)$ is the next node to be traversed to.

2.2 Our implementation

To implement MCTS in our project, we consulted and adapted boilerplate code from the online tutorial at [1]. We implemented a node data structure for the MCTS tree. Each node, representing a game state, has instance variables that represent:

1. The state of the chess board's wall positions
2. The positions of the student and adversary agents
3. The node's parent and children

4. The number of games that have been won, lost, or tied

5. The list of potential actions to be taken from the node that have not yet been simulated and turned into child nodes

Upon being launched, the student agent creates a root node using the game state passed to it by the game engine, and calls the root node's `find_best_move` function. The `find_best_move` function operates in a loop until a time limit of 1.8 seconds is reached. In each iteration, our tree policy selects a node to simulate. Then, our default policy simulates the game to the end by starting from that node and taking random moves on behalf of both players until the end of the game is reached. The data from the simulation is then back-propagated throughout the tree. Once the time limit is reached, the child with the highest ratio of wins to total games simulated is returned.

Our tree policy expands the first entry in the list of possible moves if the node currently being considered has not yet been fully expanded. Otherwise, it applies the UCT method, using Equation (1) with $c = 0.1$ to traverse to the best child until it reaches a leaf node, upon which it returns the leaf node, or it reaches a node that has not yet been fully expanded, upon which a move in its list of possible moves is expanded. When a potential move is expanded, it is taken off of the list of possible moves and a new child node is created from it. New child nodes are automatically penalized with the equivalent of 20 lost games if the player character is surrounded by 3 walls, as such moves are likely to cause the player to lose from becoming trapped.

The generation of the list of possible next actions makes use of a function derived from the `check_valid_step` function from `world.py`, applying breadth-first search to find all the positions the player can traverse to and then iterating through all possible wall positions. The list of next possible actions is generated upon the creation of a node and while games are being simulated by the default policy.

3 Quantitative performance

Both the breadth and depth of the tree policy will depend on the performance of the machine, as the program is designed to stop performing MCTS—and therefore stop adding new tree nodes—when the time limit is reached.

Depth: For a game board of size 12 and a time limit of 1.8 seconds, the maximum empirically-observed depth for the search tree was 1 (two layers,

including root node).

Breadth: The maximum empirically-observed breath (here, measured by the number of nodes at the depth with the most nodes) with size 12 board was 13.

Scaling with board size: We recorded the average maximum depth and average maximum breadth for board sizes from 6 to 12. The results are shown in Table 1 and Figure 1.

Board size	6	7	8	9	10	11	12
Breadth	104.71	99.08	43.17	36.25	19.29	13	8.6
Depth	2.86	2.42	1.5	1.17	1	1	1

Table 1: Average maximum depth and maximum breadth for various board sizes

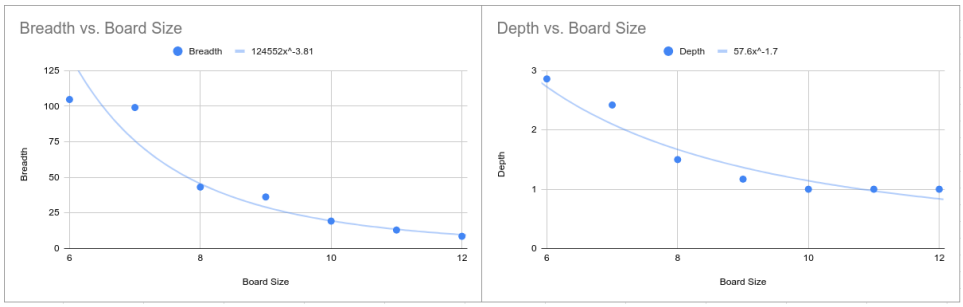

Figure 1: Plots of average maximum breadth and depth for board sizes from 6 to 12, based on data in Table 1

Trend lines suggest rough big-O estimates of $O(\frac{1}{n^{3.81}})$ for breadth and $O(\frac{1}{n^{1.7}})$ for depth. Board size appears to affect breadth more than it does depth.

Impact of heuristics: In our project, we employed the heuristic of heavily penalizing nodes in which the player is surrounded by walls on 3 sides. At board size 6, this yielded an average maximum breadth of 97.67 and depth of 2.5. At board size 12, the average maximum breadth was 14 and the average maximum depth was still 1. Removing the heuristic seems to have increased the breadth at board size 12. This may be because not culling obviously poor choices caused the player to enter poor positions from which simulated games tend to end earlier, allowing for higher breadth.

Predicted win rate: We observed a win rate of 100% against the random agent after autoplaying 50 games; thus, we predict a win rate of

close to 100% against the random agent. To predict the win rate against an average human player, we played 5 games against our student agent and won all of them; thus, against a human player, the likely win rate is $\frac{0+1}{5+3} = 12.5\%$ after Laplace smoothing for the 3 outcomes of win, loss, and tie (see L12, Slide 22). To predict the win rate against other student agents, we made our student agent autoplay against an improved version of the agent described in Section 5 which was modified to use the same method for finding possible next moves as our current student agent. Our student agent won 80% of the time after autoplaying 10 games.

4 Advantages and disadvantages

4.1 Advantages

Monte Carlo Tree Search, compared to other algorithms like minimax, excels in environments with high branching factors and limited resources. Our implementation focuses on subtrees that have more promising moves by using Upper Confidence Trees (UCT) method. This approach minimizes search space while still delivering optimal solutions within the given time limit of 2-seconds, thus eliminating the problem of the games high branching factor. MCTS is also advantageous since we don't have to know the game very well to be able to simulate the game and achieve good results. We only need to be able to implement the game mechanics and simulate legal game play from some state to finish. Indeed, our only heuristic is trivial, heavily penalizing a move if it causes the agent to be surrounded by three walls. This would not be the case with other algorithms, like minimax, which would require an evaluation function to cut-off searching before the time or memory limitations are exceeded. Good evaluation functions usually require extensive knowledge of the game.

4.2 Disadvantages

A significant disadvantage lies in the randomized nature of the simulation step. Our default policy simply picks a random move to play out of the possible moves the agent can make. This may cause our agent to overlook crucial moves in certain scenarios, especially against skilled human opponents. Skilled human opponents may see more subtle ways to play a certain scenario through intricate strategies built through extensive experience with the game. During simulation, our agent would likely overlook these more complex strategies due to the random default policy during simulation. Ad-

ditionally, if the c parameter of the UCT equation is not picked well, this would result in less exploration time for certain nodes. These nodes could be very significant when expanded much deeper in the tree, however UCT may choose not to explore them depending on the c parameter.

5 Other approaches

In a previous iteration of the project, we attempted to use an agent with a look-ahead level of only 2: the agent generates the list of possible moves and evaluates each potential move a using the heuristic of the number of possible moves starting from a, cutting evaluation short when a runtime of 1.5 seconds is reached. The heuristic has some value because, in the end game, the size of a player's territory corresponds to the number of positions that it can move to if step limits are ignored.

The previous implementation had a low look-ahead and therefore was unable to account for the long-term effects of moves. It also had a less performant way of finding possible next moves: instead of running breadth-first search once to find all the possible moves, each position within the max step radius was checked for validity by calling the breadth-first-search-based `check_valid_step` function a separate time, leading to redundancy. The implementation was better than the random agent, beating it roughly 90% of the time, but performed poorly compared to the current MCTS implementation.

6 Potential improvements

Potential improvements to our MCTS Agent could be to look at the effect of the c parameter on win-rate. This parameter dictates how much our agent explores the game tree, expanding nodes that do not necessarily have the highest win-rate, but haven't been simulated as much. Our approach currently favours exploitation heavily, that is, the c parameter is 0.1. In the future, we would make a plot of win-rate versus c parameter and try to find an optimal c parameter that balances exploitation and exploration. Additionally, since a major weakness of our agent is the random default policy, we could spend more time understanding the game itself to gather good heuristics. We could then use these heuristics in our default policy for move selection. Furthermore, we could also implement Rapid Action-Value Estimation (RAVE) which would estimate the value of playing a move immediately by looking at all the simulations where the move occurs. RAVE

assumes that only the move itself matters, which simplifies the state space and thereby requiring fewer simulations to get good results. (L8, Slides 61-62)

References

[1] Monte carlo tree search (mcts) algorithm tutorial and it's explanation with python code. [Online]. Available: https://ai-boson.github.io/mcts/