Arman Shroff-Mehrabadi (arman.shroff-mehrabadi@mail.mcgill.ca)
And Nafis Rahman (nafis.rahman@mail.mcgill.ca)

Our Approach and Justification

We chose to implement a Monte-Carlo Tree Search (MCTS) algorithm for our AI agent and did not attempt any other approaches. Due to the large number of possible moves that can be played from a given position, resulting in a very large game tree, we ruled out using the minimax algorithm or similar methods. The computing and timing restrictions of two seconds per turn had us concerned that minimax would not be able to predict the effectiveness of future moves. Furthermore, as we do not have a particularly good understanding of good strategic play for *Coliseum Survival*, developing an effective evaluation function, which is required for minimax, and/or other heuristics would have been time-consuming and error-prone. Thus, we decided that MCTS would be a much better method as it is domain-independent, meaning it does not require any prior knowledge about what a “good” move is. As well, since MCTS continuously updates the best move as it gains more knowledge over time, we could incorporate the timing constraints into our implementation, while still yielding useful results.

Our implementation begins first by mapping out the next possible moves our agent can make from a given game state. Using the current state of the barriers, the position of the two players, and the maximum step length, a breadth-first search is performed to find all valid moves our agent can make in that turn. The search begins from the initial position of our agent in the grid (i.e. the game board), moving outwards to consider all possible valid directions (i.e. those not blocked by a barrier). Each valid move is represented as a node in the game tree, in which we update the position of the agent that moved, as well as the position of the newly placed barrier. Parent nodes

represent moves that must be made before the subsequent move (i.e. the children moves) can be made. The root node represents the current state of the game, with its successor nodes representing possible moves that can occur in the following turn. The children have the successor nodes represent the possible moves that can occur if each child node is picked. As there can be over a hundred possible valid moves from a given state, the number of nodes in the game tree is roughly 100^d, where d is the depth of the tree. Within each node the barrier and player positions are stored, as well as information used by MCTS to pick the best move.

Once the search is complete, the valid moves are returned as a list of node objects. Due to time constraints, we take a random sample of fifteen of those nodes to consider as possible next moves for our agent. We then simulate the outcome of each of those nodes by randomizing the moves picked by each agent until the simulated game is complete. At this point, the function, named simulate(), returns 1 if our agent wins and 0 for a loss or a tie, as ties and losses are grouped together in our assessment. After each simulation, the number of wins and number of simulations for the node is updated and its UCB1 score is calculated. This score (formula courtesy of Russel, Norvig 2021 shown below) is used to determine the best move for the agent to pick (i.e. the greater the score, the better the move).

$$UCB1(n) = \frac{U(n)}{N(n)} + C \times \sqrt{\frac{\log N(\textsc{Parent}(n))}{N(n)}}$$

After each of the fifteen nodes have been simulated once, we pick the node with the highest UCB1 score, expand a random sample of fifteen of their children as we did previously and simulate the child nodes. After each child node is simulated, we pass up

its number of wins and simulations to its direct ancestors (parent, grandparent, and so on) up to a depth of 1. After the results of the simulations for each of the fifteen nodes have been “passed up”, the UCB1 scores for each of the affected ancestor nodes is updated. At this point, the node in the tree with the greatest UCB1 score is then selected. This process of expanding the node with the greatest score and simulating fifteen of its children repeats until the time limit is reached. To make sure that the time limit is respected, the elapsed time since the beginning of a turn is checked at every iteration of the main loop within the simulate() function. The time check is performed within this function as we found, through some testing, that it was the slowest function by a considerable margin. Once the time limit is reached, the simulate() function returns -1, causing the main loop of the program to break and we return the position and barrier placement of the node with the greatest score at depth 1.

As stated above, we used this approach as it enabled us to yield a good (i.e. far better than random) move within the time constraints imposed, without the need for prior knowledge about the game.

Theoretical Basis

While testing our function that finds the valid moves, we discovered that, at the beginning of a game in a 12x12 board there would be roughly 200 valid moves that an agent could make. While this number would likely decrease substantially later in the game, the number of nodes in the game tree would still likely be on the order of 100^d for at least the first few moves, where d is the depth (number of subsequent moves) of the tree. This would mean that the minimax algorithm, even aided by alpha-beta pruning, would not be able to look more than a couple moves into the future. As well, it

is difficult to define a good evaluation function for moves in *Colosseum Survival* as the territory that a player occupies can only be calculated when the game is over. Thus, making comparisons between different moves mid-game is difficult.

The use of MCTS alleviates the issue of not being able to compare moves in a time-efficient and accurate manner. Simulating the result of moves by randomizing subsequent moves between the two players until the game is over can be done fairly quickly. The estimated result, based on simulations, of each possible move provides an accurate way to compare moves by classifying them based on a score calculated with the UCB1 formula (shown once more below).

$$UCB1(n) = \frac{U(n)}{N(n)} + C \times \sqrt{\frac{\log N(\textsc{Parent}(n))}{N(n)}}$$

The first term balances the exploitation of moves with good average utility (win percentage) with the exploration of moves, represented by the second term, that have not been simulated as much as the others.

This enables us to more optimally use our limited time to better compare the moves. As well, by expanding the promising nodes representing promising moves and simulating the subsequent moves, we are able to consider the consequences of the initial moves, which should yield more accurate results in selecting a good move. MCTS is well suited for such a situation as simulating moves can be done cheaply. Also talk about the difficulty of a good evaluation function.

Advantages and disadvantages of our Approach

The main advantage of our approach to designing our agent is that it always carries out a logical, valid move within the time and memory restrictions. We went

through exhaustive testing to make sure we never went over the two second per turn restriction. At the same time, we made sure that the program did not stop running prematurely so that we could take advantage of the full length of time to run more simulations and get a better move. As a result, we can guarantee that our agent will always respect the time limit without having to resort to making random, uninformed moves.

As for disadvantages, we have to randomly pick a subset of moves to run simulations on from our list of all possible valid moves due to timing constraints. Without any random selection, there would usually be around 200 possible moves for the first turn for a 12x12 board, which results in each node in the game tree having around 200 children. This would result in too many nodes to run a useful amount of simulations per node. by selecting a subset of moves to simulate, we are ignoring a large number of other possible moves. Some of these ignored moves may have resulted in better outcomes than the moves in our selection. Thus, resulting in our agent choosing to make a suboptimal decision.

An additional disadvantage of our implementation was discovered while testing our helper function that found all the possible moves. About 1% of the moves it returned were invalid, forcing us to verify that each move it returned was actually valid. This costs us some time, but in the long run, stops us from making invalid moves.

Future Improvements

As previously mentioned, one of the shortcomings of our agent at the moment is that it randomly picks only 15 moves from a list of all possible moves before carrying out MCTS on a game tree containing the 15 chosen moves. This is currently leading to our

agent making suboptimal moves as a large number of possible moves are being ignored. We could significantly improve our agent by implementing a heuristic function that would determine the best moves to pick. The heuristic function would have to take into account player and opposition location of each possible move to maximize the area our agent will enclose through placing barriers. This would lead to picking moves more suited to giving our agent a better chance at winning each game, as well as eliminating redundant and suboptimal moves. As a result, we would be getting rid of the randomness associated with our agent at the moment.

In addition, we could also store results from each game played and the moves made by our agent during those games. We would use this data to aid any future moves our agent will make in future games, thus implementing a form of learning. Moves that resulted in a positive outcome would be encouraged in future games while moves made by our agent that led to losses would be avoided in future games. However, as we are not allowed to read or write files for this project nor implement any form of learning, this would be something we would have to implement outside of the context of this project.