COMP 424 Final Project Game: *Colosseum Survival*

Author: Arinc Utku Ayduran Student ID:260830748 (individual submition)

1. Summary

In this project, I implemented the Monte Carlo Tree Search (MCTS) algorithm with Upper Confidence Bounds applied to Trees (UCT) selection policy. I mainly chose this algorithm because this game has a high level of uncertainty. The initial positions of players and barriers, board size, and maximum number of moves are all randomized. In addition, the moves that players make are mostly non-deterministic, which adds to the uncertainty of the parameters of this game. Due to this uncertainty and the high branching factor of the game, implementing MCTS, which simulates games instead of relying on evaluations, was the best approach.

2. Design

My version of studenagent.py has 4 classes: StudentAgent, MCTSNode, MonteCarloTreeSearch and Simulations. In addition to the StudentAgent class, which was included in the starter code, I added 3 new classes to the agent. First, I created the MCTSNode class. The main functionality of this class is to execute the Monte Carlo Tree Search algorithm. Node keeps track of the visited nodes, its parents, and its children. Also, in this class, I created a function called calculateValue() that calculates the UCT value, which is crucial for selection. The value is calculated by checking if it has already been visited first. If it has not been visited, it will return a UCT value of 0 or else it will return a value equal to Q/N + 0.5 $\sqrt{p2\log{(parent.N)}/N}$. I used this formula because according to the textbook, to get optimal play, for most games, it's important to balance exploration (selecting states that haven't been explored much) and exploitation (selecting states that have enough simulations and have a good winning percentage). The most used selection policy is the UCT (upper confidence bounds applied to trees), which ranks the moves based on the formula: U(n)/N(n)+ C*$\sqrt{\log{(parent.N)}/N}$. After creating the MCTSNode class, I set up the Simulations class. This class checks and keeps track of the play simulations in MonteCarloTreeSearch. Also, this class has several important functions that help move forward the state of the game and the overall functionality of my agent. The first function I implemented in this class was validSteps() which creates a list of valid moves to play for the current player. To generate the valid moves, I started by creating a move for every possible combination of movements in the x and y coordinates that sum up to less than or equal the maximum step allowed, along with each possible direction of placing a barrier. Then, I checked if the move was a valid move given the current state of the chess board, ensuring that no move would lead to the player passing through an existing barrier on the chessboard, no move would lead to the player trying to place a barrier at the location where one already exists, and no move would lead to the player trying to pass over the opposition. After creating the validSteps() function, I set up a new function called play() that is used to execute a specified move for the current player. Furthermore, there are two functions in Simulations that are copied from world.py: check_endgame and check_valid_step. check_endgame is responsible for checking if the game ends and computing the current score of the agents, and check_valid_step checks if the step the agent takes is valid (reachable and within max steps). To introduce the MCTS algorithm to my agent, I created the MonteCarloTreeSearch class. One of the most important functions in this class is the heuristic() function, because this function is used throughout the execution of the Monte Carlos Tree Search algorithm. The main functionality of this function is choosing a move that is slightly better than the random moves. First, I determined whether there was a move that would allow me to end the game. If so, I would return that move. I then created two lists that include preferable moves. The first category of moves includes those that, upon execution, leave the agent with a maximum of two barriers surrounding it. I can try to exclude the prospect that our agent may surround itself with three barriers by doing this, which could allow the opponent agent to close the fourth wall around our agent and win the game. The second list of moves includes those that deviate from the chessboard boundary by comparing the move's position to either 0 or the maximum board size of 1. By doing this, we may prevent our agent from becoming trapped against the wall and, therefore, losing the game. Lastly, we remove movements that result in a loss for the agent after the game ends. We just return a random move selection from the set of potential movements if there are no moves in the two lists that were specified. The next Monte Carlo Tree Search function I want to discuss is getNode() will choose a child node to visit after receiving the current state from Simulations and the game's root node. With a single move, the child nodes correspond to every potential attainable state from the current Simulations. This technique favors nodes that have not yet been visited and uses the heuristic function to choose which node to visit. Until it reaches a leaf without any children or a terminal node, this function iterates across the children of the nodes. In the event that it comes across a leaf node, it will produce the offspring for that node and return one of them using the heuristic function. It will just return to the terminal node if it reaches one. Furthermore, I was eventually able to finish my implementation's step function with the help of the MonteCarloTreeSearch class. Within the step function, we first verify if this is our agent's initial action. If this is the case, we will build a MonteCarloTreeSearch object with a state that represents the current chessboard position, our position, the position of our opponent, and the maximum step that may be taken. Unfortunately, for our first step we use a 20-second time constraint to run the MCTS search function. I know this violates the 2 second time limit. I will discuss this more in the disadvantages section. The move that corresponds to the child node of the root that has the highest UCT value discovered by our MCTS algorithm is then chosen as the best move of the MCTS. We use the heuristic function to break the time if more than one node has the same UCT. Lastly, we play our move locally to update the state and return the move from the function. If it's not our agent's initial move, we'll figure out what the opponent did and play that move on our current state to make sure we stay in the game. This allows us to restart with an empty node from the current state and reuse nodes that were executed during the previous iteration. We initially look for a potential move that could terminate the game and help our agent win before using our MCST search function to return the best move discovered. In such case, we win the game by returning that move.

3. Quantitative Performance

We can represent the Colosseum Survival game in tree form, where each state represents a node, and each edge connecting the nodes is a decision a player makes to achieve the goal state. The board is shaped like a m x m matrix, where m can be any value between 6 and 12, hence it would be illogical to construct the entire game tree. If I were to attempt to predict the time complexity of my agent, I would predict it as O(mkI/C), where m is the number of random children to consider per search, k is the number of parallel searches, I is the number of iterations, and C is the number of cores available.

4. Advantages

There are two things that my agent is great at. First, my agent excels at not exceeding the time limit. For the most part, all of the steps take less than 1 second to resolve, which is great for performance. Second, the player that we are representing is able to make more reliable moves because the other player does not choose arbitrary actions throughout the algorithm's simulation phase. This is mostly due to the likelihood that the other player will not choose a random move that would end the game by forcing it to lose by boxing itself into a space. This process is ensured by the heuristic function, which is clearly one of the strengths of my program.

5. Disadvantages

I think one of the weakest parts of my implementation is sacrificing win rate percentage for the advantage of balancing exploration and exploitation within the time limit. As a selection policy, my agent uses the UCT formula, which is crucial for generating simulations to take advantage of the exploration and exploitation balance. However, without UCT, the average win rate of my agent is considerably higher than with my current implementation. It is possible that implementing the same agent without the UCT selection policy might be a better implementation for this specific game, but it is not clear that sacrificing UCT for a higher win rate is really worth it. Furthermore, the first move of the game exceeds the time limit in my implementation. In my opinion, this is not a disadvantage but it violates one of the rules of the game.

6. Previous Approach

At first, I considered using the A* search algorithm, but the uncertainty of the initial positions of the players and barriers quickly changed my mind. A* search would be useful if we were to calculate the shortest path between two specific locations on the board. Unfortunately, that is not useful for the Colosseum Survival game. This game requires an approach that can deal with the uncertainty of several parameters, which is the MCTS algorithm. I also thought about using other algorithms, such as min-max pruning, but I failed to implement them. Therefore, I decided to implement the MCTS algorithm.

7. Improvements

As an improvement, we can make the search in Monte Carlo Tree Search faster. One way of doing this is by giving more importance to some nodes than others and allowing their children's nodes to be searched first to reach the goal state. However, deciding which node to pick is the real challenge. In addition, we can use implicit minimax backups to improve the performance of a game. According to a paper published by computer scientists at Cornell University, using implicit backup values leads to stronger game performance in games such as Kalah, Breakthrough, and Lines of Action. We can find a way to implement the same logic in the Colosseum Survival game. Finally, as I mentioned in the disadvantages section, the win rate can be improved by changing the selection policy.

8. References

(Artificial Intelligence: A Modern Approach, 4th US ed. by Stuart Russell and Peter Norvig) Chapter 5.4
https://arxiv.org/abs/1406.0486
https://github.com/masouduut94/MCTS-agent-python/tree/2df128f87299ff3f93fc7ac7bba507ab526257f7