COMP 424: Artificial Intelligence
Abdouallah Tahdi 260989201
Leena Jannatipour 261133420

1. Motivation:

Our student agent uses a modified implementation of the Monte-Carlo Tree search (MCTS) algorithm to find the best move to make for this Colosseum Survival game. This algorithm is known to be efficient for two player zero-sum games like this one. Our agent represents a tree where each node corresponds to a game state with barriers placed, the position of player 1 and the position of player 2. As the algorithm runs for the allowed two seconds per turn, the tree is first expanded from the input game state to generate possible next moves. Then, it runs a certain number of simulations for some of these moves to check if they lead to victories or not. These simulations are not completely random moves as we want to prioritize moves that look promising. Each of these simulations goes for at most 20 moves to prevent the simulations from running for really long times when in early game and/or when game board sizes are large. We believed that with this method, our agent can beat the random agent on a consistent basis and perform well against other student agents. The primary goal being to enclose your opponent, we believed that making aggressive moves by trying to enclose the opponent should be the way to go to win more games.

2. Implementation:

Our implementation of Monte-Carlo Tree Search relies upon a tracking heuristic, which attempts to minimize the distance between our agent and the adversary while preventing possible barrier enclosures, to optimize upon our opponent's weaknesses in the simulation phase of our agent's search. Additionally, our search algorithm selects nodes based on a tree policy that utilizes Upper Confidence Trees (UCT).

Our agent starts by initializing a Monte Carlo tree using the input game state and maxsteps. Then, the best_action function is called to find the best action the current agent can do. We go through the board and set the barriers in both directions in the chess_board to avoid passing through walls. Then, we call the monte_carlo function to generate moves, analyze and evaluate them. The Monte Carlo function we built has 3 main steps:

First, we have the expansion step where we compute all the possible moves that can be made by our agent using a breadth first search algorithm and create new nodes for each of the child nodes of the current root representing the current state of the game.

The BFS algorithm consists of a queue where possible cells that can be reached from the current position are appended at the end of the queue. We iterate until the queue is empty and we have visited all the cells that can be reached within maxsteps. To avoid adding multiple times the same moves in the child list, we use a 2d array representing the board and set the corresponding visited cells to true once they have been visited once.
Then, once all the moves have been expanded from the root node, we proceed to the next step, which is to simulate games. Since each turn is limited to two seconds for the agent, we limited the simulation and backpropagating steps to 1.8 seconds. We want to keep making simulations for as much time as possible so that when we have to pick the best move, we will have UTC values that are as accurate as possible within the time limit. We added a parameter called num_sim for the simulations that varies depending on the board size. Since we know larger board sizes tend to take significantly longer to simulate, we adjusted the number of simulations to ruin so that no matter the board size, our agent will make as many simulations as possible within the time limit to obtain valuable UCT values.

In the simulate function, we play the game out by calling the check_endgame method to make sure that the game is still not over. If yes then we return the winner of the simulation. If not, we call the best_move method so that the current player can make a good move. The best_move method is used to determine the most valuable move in the absence of finding a terminal leaf node that would end the game. Using two heuristics, chase_adversary and valid_barrier_positions, our agent attempts to advance towards the adversary and place a barrier in the direction of our adversary's position, only if this barrier position does not enclose our own agent. In the case that there are no valid_barrier_positions (there are no barriers in the direction of the adversary that would not self-enclose our agent), then best_move places the barrier at random. The same occurs for chase_adversary, if there are no advantageous moves towards the adversary, then best_move will execute a random walk.

We keep playing out the game until the game is either over or this simulation of a game is taking too long to simulate and we consider it to be a defeat. We decided to limit the number of moves to be done in simulations after we realized that games with large board sizes took a very long time to simulate and made our algorithm very inefficient.

Once the simulation step is done, we are reaching the final step of the MCTS algorithm, which is the backpropagation. It goes from the current leaf node and updates the number of times that node was visited, the number of wins making the move representing that node made and calls the compute_uct function who is responsible to calculate the new uct values of that node using its visited and wins parameters.

After the 1.8 seconds is over, the monte_carlo function is over. Now, we come back to the best_action function where the last step is to call the bestChild function to go through all the children of the root and compare their uct values to find which one has the best uct value so that we can pick it as our agent's next move. The new position of the agent and the direction of the barrier are then returned.

3. Agent's performance

The breadth level constantly decreases as the board size increases due to the decay of the number of simulations - 20 simulations for size 6 to 2 simulations for size 12. The breadth level attained by MCTS is highest for board size 6 and lowest for board size 12. Additionally, the depth level for each size board is based on the valid steps and barrier positions available within the maximum steps for that given state, thereby varying depending on the current game state.

The breadth factor scales constantly to the number of simulations allocated for that board size. The depth factor, by contrast, scales linearly, O(N), to the available steps and positions where N is board size and subsequent moves are reliant upon the steps allowed given a specific board size. The breadth factor remains constant throughout each game play based on the number of simulations allocated in our MCTS.

Our best_move algorithm works similarly to a heuristic evaluation function as it yields the best move and barrier position based on their respective placement algorithms, chase_adversary and valid_barrier_positions. By biasing our MCTS search towards more promising moves in the simulation phase, our agent's exploration will always be influenced advantageously. With our heuristics simply returning the best moves and not ordering them based on advantageous move priority, our MCTS will only sometimes reach the greatest depth possible and return the best move based on efficient exploration. Compared to a random-walk based MCTS, our agent achieves a much higher breadth search level and a much deeper depth search level when applicable.

Against a random agent, our MCTS agent is predicted to win roughly 96% of the time because of its heuristic-based exploration. Against a human agent, we predict our MCTS agent's win-rate to be around 20% as our evaluation function works very similarly to a commonly-used human heuristic of advancing towards the agent with aggressive attempts to enclose the adversary while avoiding self-enclosure. However, a win becomes more difficult to obtain against a human agent as the board size increases. Against other classmates' agents, the win-rates are dependent upon the level of sophistication of our adversary's strategy, but we predict that our agent will win against

opponents with less aggressive strategies (such as only advance to the middle, only avoid self-enclosure, etc.) and less exploration capabilities, but will lose against agents that employ sophisticated exploration techniques, balanced efficiently with their exploitation strategies to make an always advantageous move (i.e. less aggressive than ours to avoid corners and self-enclosure, adapts their heuristic to go towards the middle further on in the game, etc.).

4. Advantages and Disadvantages

Our approach has many advantages. First, it computes relatively efficiently a 'good' move to make by comparing many other possibilities within the time limit. MCTS is a known and efficient algorithm for two players zero sum games that tends to outperform simpler ones such as A* or BFS. Our heuristic for move selection is efficient computationally and performance wise.

Disadvantages of our approach include having to find the perfect tuning for the parameters we used such as num_sim and the number of moves allowed in a simulation. Our algorithm does not adapt to the other agents' playstyle or use optimal strategies based on the state of the game. Our agent also goes to corners sometimes and might put himself in difficult positions that a more efficient agent could take advantage of.

5. Other approaches

Before using our MCTS approach, we explored other methods that looked promising such as Alpha-Beta pruning and and A*. We looked online at videos and articles that compared their efficiency and which one to choose for which type of games.

6. Improvements

Our agent could be improved in many different ways. There are many parameters that we choose arbitrarily in our code such as the num_sim parameters that run simulations a certain number of times depending on the board size and the limit of 20 moves during each simulation of a game. These parameters were chosen based on the observed results of the autoplay tests we ran, but they are not the most efficient values for sure as we could not test all possible combinations of values. Also, our observations could have been misrepresented due to the variance of the results since the random_agent plays random moves each turn as his name indicates. Another possible improvement for our algorithm could be to use better heuristics. Although we think our agent chooses moves efficiently, there are probably times where aggressive moves being prioritized might not

be the best way to win games against other agents in the class. Adapting our agents' heuristics and play style based on the number of moves already made, the number of barriers placed on the board or on how the opponent's agent is playing is something we did not explore and could have potentially improved the efficacy of our agent. Certain functions in our code could be optimized to use more efficient data structures and save some time which could be used to make more simulations. Something else we could have tried to improve our agent was to order possible moves to be explored in a certain order based on a heuristic that helps find the moves that have higher potentials and explore them in priority. For example, moves that make your agent closer to the board should be avoided and try to remain somewhere around the center area. Another idea to be explored could have been to make our algorithm able to remember moves that they made and make it learn from its past mistakes. This way, the performance of the agent could get better as it plays more games.

References used:

https://www.youtube.com/watch?v=lhFXKNyA0QA
https://ai-boson.github.io/mcts/
https://www.geeksforgeeks.org/ml-monte-carlo-tree-search-mcts/
https://www.rebellionresearch.com/what-is-monte-carlo-tree-search-used-for