Report of final project COMP424

Hao Chen He (260917762) and Cindy Hu (260800514)

April 8th 2022

1 Introduction

For the final project, we are tasked to reproduce a game of "Colosseum Survival" using our own heuristics and algorithms to create an agent using the code provided.

2 Technical approach

2.1 Motivation for the approach

Our game is composed of two players. For the purpose of this project, the players would be composed of our own agent and a random agent (or another team's agent) which will be provided. Therefore, we do not have much information about our opponent and the moves that they plan to do.

Since it is a game, our first thought was to use an algorithm that is commonly used for games and that allowed us to predict optimal moves for our own agent and for our opponent's moves as well. Our project, "Colosseum Survival", is a 2-player, turn-taking, perfect information, deterministic game. Therefore, this game can be modelled as a search problem. We have perfect information because, as the game starts, our agent receives information about the board. It receives the board's size, the starting positions of our agent and of the adversary agent as well as which positions already have barriers in a certain direction on the board.

Indeed, our game has multiple variables we can work with, such as, previously mentioned, the size of the board (which is M x M, so always a square), the positions of the agents on the board (using coordinates to describe the positions), the four different directions that our agent can travel to (left, up, right, down) and the four different directions where barriers can be put by the agents (also left, up, right, down).

Because our goal is to model a search problem, the first thing we wanted to figure out is how to create or implement a search algorithm from this game and to define our utility, which is the way we measure how optimal a step our agent wants to take is. Therefore, our original idea was to implement the

Monte-Carlo Tree Search algorithm then to apply a Minimax algorithm with alpha-beta pruning to allow our agent to play the game optimally.

We were considering these algorithms, because they all allow us to search through many possibilities of game play to find a sequence of moves that will allow us to win most of the times. Indeed, we know that the Monte Carlo Tree Search will allow us to try out different possibilities because our agent is not yet aware what the optimal step is. So we only need it to choose one sequence of moves that works for now, since there are multiple moves that can win. As long as our agent has more squares enclosed, it is considered the winner. However, of course, if we want an optimal result, then we would want to win with the most squares enclosed as possible compared to our opponent. Minimax is an algorithm that allows our agent to maximize its moves while also minimizing the moves the opponent makes. Alpha-beta pruning is an optimization algorithm that will allow us to cut down the computation time to find the optimal path for our agent, because the Minimax algorithm can take a while to compute, especially given the situation of our game. Indeed, the reason why we wanted to begin our search of an optimal solution for this game with a Monte-Carlo Tree Search was because our game has multiple options of moves it can make. Given each possibility of an initial board, there is K maximum numbers of steps our agent could take, where $K = \lfloor \frac{M+1}{2} \rfloor$. For each of those steps, there are four possible directions where we can put barriers. Therefore, the first move would already have at least $Kx4$ combinations, each with different ways of making our agent move, meaning that our agent can decide, for instance, that if we had $K = 4$, to move 2 blocks down, then 2 blocks right, or 2 blocks up, then 2 blocks left given that those moves are possible, and many more. This creates a very big search tree that needs to be implemented, therefore we thought that only using the Minimax algorithm would not be ideal because the computation time would be too long.

2.2 Our actual implementation

Due to multiple difficulties, we only managed to implement Minimax algorithm for our agent. More specifically, one-step Minimax is used for our agent as trying to make our agent predict more layers has failed.

When Minimax is applied to our agent, we first retrieve the set of possible positions which the player is allowed to reach. This is done through BFS (Breadth First Search) as this minimizes repetitions, more specifically, of landing on a step that was previously landed compared to DFS. We confirmed this by testing it and implementing DFS originally. We then found that using DFS would cause more repetitions than BFS. Throughout the process, we have to verify whether there is a wall in the direction we are moving, whether our opponent is at the new position our agent is trying to travel to, and whether it is within the boundary of the board. If any of these happens, that direction would not be a possible next step, so a different next step must be taken.

For each next step which our agent allowed to land on, there are at most four possible directions which we can place a wall - left, up, right, down. For

each of those four directions, we generate a new board configuration (which is a possible next state) by taking into account the fact that each wall is shared by two neighboring positions. Indeed, say, (0,0) shares its right side barrier with (0,1), indeed the right side barrier of (0,0) is the same as the left side barrier of (0,1). Therefore, each barrier has an opposite barrier we must take into account. For each board configuration, we would apply our heuristic and get a utility value from that heuristic. The heuristic is to run BFS for both players by alternating the player depending on their turn. To specify, each will never have two consecutive turns. Each player plays one turn at a time. At each turn, the player expands their BFS tree. When a player lands on a position, that player appropriates all the possible positions that it could move next. We chose to use the positions instead of making them appropriate a specific location and direction because, as long as a player is able to land on a specific position, it is up to that player to decide which direction to place a wall, so the player that appropriates more positions is more at an advantage. A difference between the number of positions that each player can appropriate is computed. The state which favors the current player's appropriation the most is the best next state, i.e. the best next position and direction to be chosen. We also considered using the number of next possible moves for each player to create a two layer Minimax and taking that difference as a decision heuristic, but running a BFS through the entire board would take into account the locations of all walls as opposed to only the walls that are close to the player in our one layer Minimax, which seems to be a superior strategy.

We also did not use any code from other sources. All of our code was implemented from scratch.

3 Pros and cons of the approach

The pros of our approach is that we are able to find a good approach to the problem as we have tested our code by making it play against the provided random agent multiple times and found that we can indeed win very often. We have optimized a search to a good solution through our algorithm rather quickly as our model does not take long to find a move to make, indeed, our agent saves computational time and is guaranteed to not exceed the limit of 2 seconds and being forced to do a random walk.

The cons of our approach is that we cannot guarantee that our agent will win at all times. Indeed, our algorithm does not allow us to always win with the optimal solution, which is when our opponent loses and encloses itself in. For example, if the size of the board is 6x6, then we would have a total of 36 blocks. The optimal win for our agent would be that our opponent only has 1 block and is surrounded by barriers(so, it has enclosed itself in), while our agent has 35 blocks. Another con is that our algorithm might be very slow as the number of possibilities increase for testing purposes. Therefore, our implementation might be too slow at finding a good sequence of moves than other algorithm options we could have used. We are also only able to make one future prediction, so

playing against an agent that can predict several predictions ahead would put us at a disadvantage. We are also not optimizing our algorithm with a more elaborate heuristic. We will likely lose against a player that has taken more strategies into considerations when writing their heuristics.

4 Improvements

Due to lack of time on our part, we did not manage to implement our whole idea. Our original idea was to implement Monte Carlo Tree Search that will allow us to find all the winning solutions first and give us a pretty good idea as to which path was optimal. Then on those winning moves, we would have applied a Minimax algorithm to find the optimal one while assuming that our opponent would also try to optimize their own moves. To make that process faster, we would have also implemented an alpha-beta pruning algorithm that will allow us to only test out a couple of moves instead of all of them. If we had a bit more ease at implementing our code and having used a bit more time, this is what we would have probably done as we believe that this would have given us an agent that will guarantee wins no matter what the opposing agent decides to do. However, this idea was quite a challenge for us since:

- We were not sure about how efficient it would be to apply Minimax only at the end of the Monte Carlo Tree Search algorithm implementation on our game because it would put us at a disadvantage against other agents that optimize their moves at each steps.
- We were unsure if that would allow us to respect the time limit of 2 seconds per decision.

We also confirmed this idea by finding a source that claims it is possible to embed shallow Minimax searches into a Monte Carlo Tree Search Framework (https://link.springer.com/chapter/10.1007/978-3-319-14923-3_4).

Another idea we had was to start our search by using Minimax, and then apply Monte Carlo Tree Search on the two best next states for each player depending on who's turn it is. We attempted this idea, but unfortunately, our code failed. If this had succeeded, it may have lead to a better optimized result since we would have been able to make relatively more accurate winning predictions in the point of view of our agent, and this method would have taken into account how the opponent will minimize our own utility.

One last idea was to simply use Monte Carlo Tree Search with pure random explorations. however, not only is this method computationally expensive, this will also make our agent easily lose against a Minimax player. Indeed, if we found ourselves in a situation where our agent is in a position that looks for a next move that can be played, but is not necessarily optimal, the worst case would be that the next move is playable but encloses us in before our opponent loses. So, even though that next step was possible, it would actually make us lose faster because we only did a random search instead of finding the best move.