424 Final Report

Zeyu Li, Barry Li

April 2022

1 Introduction

This report serves as an explanation, description, and analysis of our team's solution for the project of solving Colosseum Survival. This report is dedicated to showing how the solution is developed and enriched from scratch, some obstacles we encountered during the process, and most importantly, some future improvements we can make to create a more scientific and precise solution to such problems.

2 How the program works and the motivation

When developing a game-playing agent, the first and foremost decision we have to make is how the agent makes its decision. We learned there are three major approaches to choose from in the class. The first and most apparent approach is the Minimax algorithm. However, it's a sub optimal solution in two ways; first, since we are playing against another student's agent or the random agent, there is no guarantee the opponent is optimal. Such a condition leads to the minimax algorithm not always coming up with the optimal solution against our opponent. Second, the time complexity for such an algorithm is unheard of, given the time constraint of 30 seconds for the first step and 2 seconds for every latter move. We doubt whether it can process all the steps in time. After further exploring, we found for a 12 x 12 board, there are, on average, 200 legal moves for the first move, which further confirmed our assumption. The second approach is the alpha beta-pruning approach. Like the Minimax, it will be sub optimal to play against a non-optimal opponent. However, the time required for such an algorithm is much faster than the first despite having the same worst-case complexity. The last approach is the Monte-Carlo Search Tree, which we took a long time to get past. Through research, the Monte-Carlo tree search does not always converge to Minimax in non-Monte Carlo perfect games.

Through careful evaluation, we chose the alpha-beta pruning approach to be the algorithm to determine which move the agent should make next. Before implementing the alpha-beta algorithm, several helpers functions such as set barriers, check end games, check reachable, and who win are copied and

re-implemented from the world-class provided. We made notable changes for the check reachable function in that this function no longer checks if placing a barrier at a specific position is valid. Instead, it computes whether a board position can be reached given a chessboard, both position of the agent moving and the other agent, and the max steps that the agent can take. We also made the return value of determining who wins from 0 corresponding to losing, 1 corresponding to winning and -1 corresponding to tying to 0 corresponding to tying and -1 corresponding to losing since we feel such changes would be more natural and intuitive. After implementing the required helper, the simple alpha-beta pruning algorithm and some speed-up techniques we came up with (which will be expanded more in the other approaches section), we decided to run our code for the first time. However, the time it requires is way too much; it can only make its first move within the frame of time needed for the 3x3 board. Then we realize only evaluating at the end when the game is officially over is a flawed approach, and we need a heuristic for evaluating a board state when the game is not necessarily over.

Then, we brainstormed several possible heuristics for computing intermediate board states. After voting and vetoing the ones that are too complicated or don't work, it boils down to 2 possible ideas. The first one is if we consider a straight line with the most barriers between our agent and the opponent separating the board. This line will divide two agents into two different rectangular regions. Then the board state can be evaluated by the algorithm as to who has more blocks within its region. The other idea is if we consider a line that can zig-zag with the most barriers that can help us win the game by one block or two blocks, depending on the board's shape. Then the board state can be evaluated by the algorithm as to how much we are contributing to the line after some moves. We expanded our discussion on the second option. We think the agent should update the line after some moves. We even went on to write several helpers. However, both time and code complexity seem too much for us. Thus we took the first approach, and we not only added several functionalities and perfected the value we should assign for the board state.

First and foremost, we need to solve what if there are two straight lines in between the agents trying to have the most amount of the barriers. The solution we come up with it is to consider them separately. Such a solution led to another problem we need to solve: what value we need to assign for each move. We decided to choose the solution due to the net win among all lines divided by the number of lines between the agents. Such a solution will land us between 1 and -1, corresponding to our updated version of deciding who wins. We separated the above implementation into two methods: a heuristic helper function to return the lines between the agents and a heuristic computation function that computes the value of the intermediate move.

Lastly, it is time to utilize those functions in an appropriate logical order to compute a step we should be taking under the function name step. The logic is

simple once all the helpers are proven to be correct. First, we compute all the unique moves in the given board state and order all the moves by randomizing them to mix in a bit of taste of Monte-Carlo Search. Then we pass in the board state after each move, and with the help of alpha-beta pruning, it will return the best move right now according to our heuristic. To avoid going over the time limit, we also limit the depth and width for different board sizes under the function ltpo.

3 Theoretical basis of the approach

As mentioned in the previous part, the algorithm we based on is Minimax search with Alpha beta pruning. The draw back is very obvious, which means it is still expensive and time consuming, and relies on a reasonable evaluation function to compute the next step. So we have to first find a evaluation function that is efficient enough to compute the score, the heuristic of each future step, so that we can decide what to do.

The Minimax algorithm, backtracking algorithm that is used in decision making to find the optimal move, assuming the opponent always act optimally, which is also known as to minimize our score (minimizer), and ourselves are acting like the maximizer (try to find the max score that we can have between different decisions).

As we mentioned before that the Minimax algorithm has a good performance but it requires a lot of time to compute, we introduced the alpha-eta-pruning method, which can improve the speed by deleting the useless but possible moves as they would not improve the current performance but will cost a huge amount of time to compute. Through deleting these points by alpha beta, we can keep the useful steps, possible moves that are optimal and at the same time reduce the running time for each single step.

To make this alpha-beta method work, we should fit the algorithm with the game using some helper functions. First we should check the possible moves for the agent, using the function "heuristic computation", which is also known as the evaluation function. It decides whether each move is optimal enough, or which step is better in the current situation. As described in the previous section of the report, it assumes the board is divided in two parts, one for our agent and one for the opponent. Our move should make our part larger,, which is considered to have a positive heuristic, otherwise it will be set to negative. It's the same as fill in the values for the minimax tree, to know what will happen, what will the score be if we choose our move.

In order to find the inputs for the heuristic functions, we use "uniquemoves" to compute each next move we can get, and use "ordermoves" to decide which

move will be the first to compute. Sometimes there could be a case that a block is near to the agent but according to the max step restriction and barriers it may not be reachable in one single round. Under this circumstances, we introduced the "check reachable" function to find if the move is possible in the next round. So adjacent blocks can be checked in order by the "uniquemoves" function to insure that we don't miss any chance of winning.

After all above computations we can now find required paths, and at the same time they are optimal enough for each specific situation. The only remaining problem would be that the computing time is too long for larger boards, such as 12 * 12 game board. If we insist to compute the first 50 choices with a depth of 5 (expand the nodes to five more moves each) will cost several hours for just one move. As the time limit is so much shorter than the computation time required, the most direct way to have an improvement would be introducing Monte Carlo Tree Search and Iterative deepening Search to reduce the blocks we explore so the effort for calculation highly decreases. This is what our "ltpo" function does. For each different board size, we have a different computation effort adapted specifically to that size. So the agent can decide the step much faster than before, and it is ensured to stop within two seconds.

4 Summary of the advantages and disadvantages

The biggest advantage would be that we could find the optimal solution under each circumstances. The minimax algorithm is always optimal when enough information is given. Although the opponent may not want to minimize our score (just like the random agent), we can still find a preferred way to execute. When competing with the random agent, the win rate could get high up to 99 percent without any targeted strategy. It can also fit into different board size through the "ltpo" function, so whatever the board size is, the win rate keeps almost constant and high. And if we want to have some further improvement, the code is managed into an integrated way to help us get an easier access to the main "step" function, just plug-in new helper functions could increase the accuracy.

In contrast, the same as all other alpha-beta pruning, the time is hard to control. Even if we have examined the time for several hundreds of times, it could still get time out if the criteria is set to two seconds, which is so strict for a program, especially when the boards are large. Maybe it can perform better on machines with a more advanced CPU. There might be other heuristics we can use that are faster and more accurate than the current one. Another thing is that the 30 seconds for the first step is not fully used, as we mentioned before that the first step should compute a much larger amount of data than the later steps in optimal.

5 Other approaches

As mentioned in part two, "How the program works and motivation." We noted that the heuristic approach was never the first approach we came up with. First, we considered the dynamic programming approaches since we want to take advantage of the entire 30 seconds period for the first move. We considered building a tree and traversing it within the first 30 seconds and having each node store information about their child, their move, their current board state, their value and their parents. The logic is as follows: during the first step, our agent will build the tree accordingly and make alpha-beta pruning approaches when the layer is max. Then during each later move, we will go to the board state that the opponent chose and then choose the child with the most value. However, this requires not to do any pruning when the layer is min. To compensate for not pruning the min, we will prune the rest of the tree if one of our max nodes results in the value 1. The reasons are as follows: first, after traversing the tree and storing all the values, all the latter moves will only choose the best value among its child. Secondly, we do not prune the min layer to guarantee that we do not prune a possible move from the opponent even though it can be sub optimal; as we mentioned earlier, there is no guarantee of the optimality of the opponent. Third, we prune the rest of the tree if a max nodes result in the value 1 since value 1 within our game evaluation is a win, and there is no way to get better than that. However, the approaches required us to traverse the tree two times during the initial move, once building and once assigning value, and we had to check all the possibilities for the min move. Although 30 seconds seems a lot, this massive amount of computations is nowhere close to the time limit. So these approaches were forced to be abandoned.

We also tried an iterative deepening approach when computing unique moves. This was due to the large computations for the 12x12 board, and we found that the most time-consuming function was this "unique moves" function. So we decided to take another approach by doing an iterative deepening search. However, the result is not ideal. It took more time, and it was again abandoned in the final moments.

6 How to improve further?

As mentioned in the disadvantage part, to help solving the time consuming problem, it might get faster on a machine that performs well. Within the same condition, we can try to use programming techniques to limit the time for each function, for example change the recursion functions to iterative functions to reduce cost.

Utilizing the first thirty seconds is essential for further calculation, so it will be optimal to construct a tree and use the dynamic programming approaches as we mentioned before. However, the problem lies in what if the opponent

chooses a move that we did not consider? Although we failed to explore further in this direction due to the time constraint, we still believe this idea to be valid, and with the help of more time and going more in-depth with the problem, we think a solution can become up.

Adding another heuristic to an already existing one is also an excellent way to explore this problem. One of them coming up to our mind is evaluating the agents by the position, where the closer the corner and walls, the lower the score. Mixing it with our original heuristic, this can work for the case when the agent goes to the side and is blocked by the barriers, which can solve some of the exceptional cases we are facing during calculation.