Artificially Dense Final Project Report, McGill University, W2022 COMP424

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

April 8th 2022

1 Introduction

The purpose of this project was to implement an AI algorithm for the agent of a large-scale game called "Colosseum Survival". With the rise of AI in different domains, it has grown in popularity in the gaming sector given its ability to make real-time and dynamic decisions in complex environments (https://www.researchgate.net/publication/220061084_Artificial_Intelligence_for_Computer_Games). Machine learning techniques further improve the decision making processes by modeling player behaviors to enhance gaming experience (https://link.springer.com/chapter/10.1007/978-0-387-09701-5_1).

For the purpose of the project, the game's rules have been studied by playing several games with different board sizes, random wall configurations, and random but symmetrical initial states of the agents and barriers before writing the code. Our implementation includes a 1-layer Minimax algorithm, with a heuristic supported by the breadth-first search (BFS) algorithm. It was tested to beat the random player with 99% to 100% winning rate and an average run time per game of less than 1 second.

2 Technical Approach

2.1 Motivation

To come up with our approach, we started by considering the characteristics of our game:

- Zero-sum: One player's gain is equivalent to the opponent's loss;
- Game has adversarial agents: The two players are playing against each other, where each player is attempting to maximize their own utility function and minimize the opponent's utility;
- Turn-taking: Each player takes alternative turns, so each player cannot play two times consecutively;
- Deterministic: State changes are fully determined by each player's move;
- Perfect information: Players are able to observe all information about a state of a game, including the board's size, the starting positions of the agents and barriers;
- Very large search space: Each player have a set of possible positions to land on, and each position has a set of directions where a barrier may be placed.

We also considered the multiple variables/information which were at our disposition, including:

- Size of the board, which is M x M, where M = 6 to 12;
- Positions of the agents on the board, described by coordinates;
- Possible directions that our agent can travel to (left, up, right, down);
- Possible directions where barriers can be put by the agents (left, up, right, down).

Given the above considerations, Minimax was thought to be an algorithm that satisfies the requirements of the game. In fact, Minimax is an algorithm that is able to find the optimal next moves for an agent depending on a certain utility function while also minimizing the utility value of its opponent. Each player will attempt to maximize their own utility value. The optimal path for the current player is the one that maximizes its utility value from the simulation. However, this optimal solution requires high computational resources, which is a crucial consideration given the decision making time constraint of the game.

Indeed, the search space of our game is very large, which, in turn, requires a long search time if multiple-layer Minimax was implemented. Given an arbitrary initial board, the rules of the game dictate that the agents may take at most K steps where $K = \lfloor \frac{M+1}{2} \rfloor$. For each of those steps, there are at most four possible directions where we can put barriers. Therefore, the first set of moves would already have at least $Kx4$ combinations if it was moving in a straight line towards the same direction. However, there are multiple directions and paths that the agent may take within K steps: 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, among other possibilities. This creates a very big search space that needs to be parsed through. Therefore, we decided to shrink the size of the Minimax search to 1 layer, such that the players are always making the optimal next move given a heuristic, which we designed such that it takes the presence of all walls into account, as described in the next section.

2.2 Actual Implementation

The algorithm implemented for our agent was the one-step Minimax with a heuristic.

When Minimax is applied, the set of possible positions which the player is allowed to reach is first retrieved. This is implemented through BFS (breadth first search), as this search method minimizes the number of repetitions for each searching step which prevents it from searching a step that has already been searched. We used BFS instead of DFS because DFS actually does not ignore repetitions. This fact was confirmed by actually implementing DFS first in our code and observing its search path during the implementation. Throughout the BFS process, the presence of a wall or barrier or board boundary in the targeted direction or blocking our way towards it was verified. We also verified whether our opponent was in a neighbouring step of the targeted direction or if we were going to accidentally lend where our opponent happened to be. If any of these applied, that future direction would not be a possible next step or place to put a barrier or even travel to.

For each next step that our agent is allowed to land on, there are at most four possible directions which we can place a wall - left, up, right or down. For each of those four directions, a new board configuration (which would represent the possible next state) is generated by taking into account the fact that each wall is shared by two neighboring positions. For example, (0,0) shares its right side barrier with the left side barrier of (0,1).

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 depending on whose turn it is. Since each player plays one move at a time, the players take turns to expand their BFS tree. When a player lands on a position, that player appropriates all the possible positions that it could possibly move to next. We chose to use the positions only instead of making them appropriate a specific combination of locations and directions 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. In other words, the player that appropriates more positions is more at an advantage. To find the optimal position, a difference between the number of positions that each player can appropriate is then 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 put the barrier at to be chosen. In general, this heuristic takes into account the position of all barriers on the board and the relative positions of both agents.

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

3 Actual Implementation Pros and Cons

3.1 The Pros

The pros of our approach is that we were able to find an optimal next move for our agent based from our heuristic. This strategy was sufficient to win against a random player for most of the tries we did. The algorithm is also optimized such that it takes less than 1 second per game, which saves extensive computational time and is guaranteed to not exceed the limit of 2 seconds and being forced to do a random walk.

3.2 The Cons

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 there would be a total of 36 blocks. The optimal win for our agent would be that our opponent only contains 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 (and board size) increases for testing purposes. Therefore, our implementation might be too slow at finding a good sequence of moves compared to 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 Other Approaches Considered

Our original idea was to implement Monte Carlo Tree Search (MCTS), which would allow us to identify the random moves that would more likely lead our agent to a win. MCTS is known to work well with larger search spaces, and that algorithm would avoid the high computational requirements of Minimax, which is expensive even after implementing alpha-beta pruning, especially since our game has a very big search space. Then, on those good random moves generated, we would have applied the Minimax algorithm on all the randomly generated winning paths to find the optimal ending move. 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. This combination was considered since both algorithm have their own advantage and disadvantages which we would like to benefit from and mitigate. However, this idea was quite a challenge for us since:

- We were unsure about the performance of only applying Minimax at the end of the MCTS algorithm on our game because it would put us at a disadvantage against other agents that optimize their moves at each step;
- We were unsure about when is a good time to switch from MCTS to Minimax;
- We were unsure if we would be able to respect the time limit of 2 seconds per decision.

We also found that it is possible to embed shallow minimax searches into MCTS framework https://link.springer.com/chapter/10.1007/978-3-319-14923-3_4. However, in order to implement MCTS, we need a selection criteria to filter the steps that are more worthwhile to explore, meaning that they would lead us to a win. This is usually achieved by training the game model. There are many ways to select a best sequence of steps that can lead us to a win. Indeed, we could have chosen the sequence that satisfied the UTC formula or the move that was the most searched throughout the MCTS process. These are all valid choices, but we were unsure which one to choose. In the absence of such a selection criteria, MCTS with only random explorations will very likely lead to a suboptimal step if the game is played against a Minimax player, and would eventually lose because it might not find the optimal sequence in time. Thus, only applying MCTS was not the optimal choice of algorithm for us.

Another idea we had was to start our search by using Minimax, and then apply MCTS on the two best next states for each player depending on the player's turn. 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 MCTS with pure random explorations. However, this will make our agent easily lose against a Minimax player. Indeed, if there is only one step we could make to save ourself from the opponent, but it is not part of the random set of next move, selecting the best random move would not help in achieving a better result: even though that next step was possible, it would actually make us lose faster because we only did a random search instead of identifying the saving move.

As for the heuristics, 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.

5 Improvements

We thought that there are multiple ways to improve our algorithms, including:

- Adding MCTS;
- Adding alpha beta pruning;
- Finding more elaborate heuristics;

- Training better models on which we base the MCTS;
- Developing some strategies to win the game, for example by using some functions that calculates the relationships between the number of walls, the locations of the wall, by taking advantage of the known info (e.g. start symmetrically with 2K barriers, and knowing that we can make at most floor((M + 1)/2) moves, among else).