COMP 424 Final Project Game: *Colosseum Survival!* Report

Flora Chai - 260888064 yuting.chai@mail.mcgill.ca
Department of Computer Science
McGill University

Charles Wetaski - 260714346 charles.wetaski@mail.mcgill.ca
Department of Mechanical Engineering
McGill University

Abstract

Games are intriguing human inventions that are passed on for generations. These pieces of art such as Go, Chess, Disentanglement Puzzle, etc, each with their own set of winning rules, are free of language and culture barriers and attract many people around the world to play or even compete formally. Naturally, people are dedicated to find strategies that maximize their wins for each game. In this report, we aim to explain the approach that we have used to implement our agent for the *Colosseum Survival!* game, where two players play against each other by putting barriers on a game board and one player wins by enclosing more area around it.

Keywords: Minimax search, Heuristics, Iterative Deepening Search, Pruning, Depth-First Search, Breadth-First Search

1. Technical Approach and its Theoretical basis

Our main approach for implementing the student agent is using the Minimax Algorithm. Minimax Algorithm is a game playing strategy when the two players in a game take turns to play and both want to maximize their own rewards. The algorithm uses depth-first search; it starts from our player's perspective in its search tree and looks at all possible moves that our player can take in the next step, and it expands each move on the branch. Then it lets our opponent expand all of their steps and continue on like this. If a move would end the game, then its result is compared against the best-move scored so far, and if it is better (from that player's perspective), then the score of that move replaces the current best score. Additionally, if the player finds a move that makes it win, then the algorithm stops checking that branch, since the best possible outcome was already found.

The possible outcomes of a check are that a move either ends the game with a win, draw, or loss. Each possibility was assigned a score as follows:

- Win = 1
- Draw = 0.5
- Loss = 0

Additionally, if the depth limit of the search is reached, a score of 0.75 is returned. The max (min) player aims to find the move which gives the maximum (minimum) score. In this

way, since our agent is the max player, our agent prefers a move which reaches the depth limit with no definite result compared to a move which guarantees a draw. In this way, the agent is betting that it is more likely to win if the game proceeds toward the undetermined position than the opponent.

Since the branching factor of the game was very high, and checking whether each position was an endgame was nontrivial (to check the end game, the entire NxN board of the new position was mapped), we found that the search could not typically search the entire game tree. Thus, a version of iterative deepening was used for the search, where, first, every possible move was searched only to a depth of 1, and then the search tree was searched to a higher depth. This allowed every move to be checked to see if it won the game. Following checking the first move, the tree was then searched to the specified depth. The depth was modified based on the board conditions. The default depth was 1, but the depth was modified based on the following conditions:

- If the board size was less than 8, the depth was increased by 1
- If it was the first turn, the depth was increased by 1
- If there were fewer than 20 candidate steps, the depth was increased by 1

Additionally, if a move had been searched to the depth limit which was indeterminate (i.e., it was not known whether the move led to a draw, loss, or win), then the depth limit was reduced so that the algorithm could search for a winning move.

For the moves to search, candidate moves were first determined by generating a list of all legal moves. This was done by modifying the method `check_valid_move` from the provided `world.py` file. The legal moves were determined by searching using the breadth-first-search from `check_valid_move` to search every possible path through the board up to the maximum move length, and returning all destination and intermediate positions. Once all reachable positions were determined, the list was iterated through and a move was stored for each direction which did not already have a wall (for each valid position).

The set of candidate moves was then pruned before applying the Minimax algorithm (and within the Minimax algorithm, before searching to new candidate moves lists at a greater depth. The two conditions which were checked in order to prune the candidate moves list were called "stupid" conditions, as they were moves that could be immediately seen to lose the game. The first stupid condition was that the agent would wall itself into a 1x1 box, instantly making itself lose the game. The second stupid condition was for the agent to move into a square which already had two borders (thus, the agent would place the third), while the adversary was in range of the fourth border. Both of these conditions allowed for pruning moves from the set of candidate moves without having to iterate through the opponent's next moves (I think I know what you mean but it seemed to me awkwardly worded... ~~the opponents set of candidate replies~~ until finding the move that wins.

If a move was determined to give a guaranteed win, the search was terminated and that move was selected. If the search depth limit was reached without determining that a move was a guaranteed win, loss, or draw, the move which returned the best result at the highest depth was selected. Therefore, a loss, draw, or undetermined outcome which is further away is preferred to one which occurs in fewer moves. The reasoning behind this is that it gives the opponent more chances to make a mistake, which would allow us to win. When

comparing two potential moves which give the same result, a heuristic called the weighted score of each potential move was defined. The weighted score of a given move was defined as:

$$\mathrm{WS} = \frac{N_{\mathrm{win}} - N_{\mathrm{loss}}}{N_{\mathrm{total}}} \tag{1}$$

Where N_{win} and N_{win} were the number of winning and losing endgame states which were found when searching that move, and N_{total} was the total number of moves searched along that node. This heuristic was neither admissible nor consistent, as the weighted score does not monotonically approach the true result of a move (assuming optimal play) as it gets closer to fully exploring the search space. In fact, a move which produces a guaranteed win (with optimal play), could (possibly) produce a negative score from this heuristic if, for each possible move from the min-player, the max-player had many moves that lead to a loss but only one which leads to a draw. This heuristic is reminiscent of the approach used in a Monte-Carlo search method, except there is no randomization involved, and no balance between exploration and exploitation. Unlike in a Monte-Carlo search, the heuristic does not affect the way that the game-tree is searched; it only effects how the agent decides between two similar moves.

2. Motivation for Technical Approach

The Colosseum Survival game clearly is a 2-player, turn-taking game where the opponent seeks to play optimally. The most basic algorithm we want to implement is the Minimax algorithm. But due to the huge branching factor of the search tree, using only the Minimax algorithm is not realistic to fit searching time within two seconds of each move. Thus we tried to prune the search space before searching by eliminating the steps that would obviously make our agent lose within 1 turn, and we implemented the iterative deepening approach to be able to be adaptive and have the chance of giving the seemingly promising moves a deeper search.

3. Pros/cons of Chosen Approach

Some pros of the approach selected is that it was relatively simple to code and that there was no risk of exceeding the maximum memory requirements. However, our implementation often cannot search very deep before hitting the move-time limit. For example, when pruning the "stupid" moves of the agent, the moves that were pruned were very limited compared to the amount of plausible steps that were left. There might not be any move that would let the agent trap itself instantly or let the opponent trap the current agent in the next step. As a result, most of the times the searching process had to be cut off by the 2-second time limit. Another con of our approach is that, since the previous searches are not stored, it has to start the search process from scratch at the beginning of every move. Despite the cons of our approach, it was still robust enough to beat the random agent more than 90% of the time in our test games.

4. Future Improvements

An improvement to the approach would be to use a breadth-first search rather than a depth-first search. A depth-first search was initially used because the standard Minimax algorithm generally expects to search all the way to the leaves of the game tree. However, in this case, due to the high branching factor of the game, this is unfeasible. Thus, in the majority of cases, the algorithm must decide on a move without being able to know the outcome. Therefore, a breadth-first search would be an improvement for two reasons:

1. The somewhat awkward rules used to determine the max-depth would not have to be used (the search could just continue until either a winning move was found, the memory limit was reached, or the time limit was reached)

2. The algorithm would lend itself well to storing the explored search tree from one move to the next.

The breadth-first search was not applied because, as one team-member had never used python before and the other team-member had only limited experience, coding even the simpler depth-first Minimax algorithm was quite time-consuming. Implementing the breadth-first search would require implementing a graph object in which the explored game-states were stored as well as a queue. Further, it would require adding checks to ensure that the memory-requirement was not exceeded. To make the algorithm high quality, it would also need to integrate a system for pruning "dead-end" paths from the graph (e.g., paths which lead to a loss, since we don't need to store all the details which make the move bad, only that the move is bad). Because we only began to understand that this search strategy would perform better than ours in the later stages of the project, and because of our limited python coding skills, we decided to stick with our initial strategy and make it perform as well as possible.

Another future improvement to the algorithm would arise from evaluating effectiveness of the weighted score heuristic. Based on its usefulness, it could either be improved (e.g., if certain situations could be identified in which the heuristic was useful or not useful), kept the same, or discarded. In order to do this, a version of the agent in which moves are selected randomly (at the level of decision making where the heuristic was applied) could be made to play against the agent in which the heuristic is used, and the results evaluated.

A third aspect that we could improve our algorithm on is that to find out moves that are much more promising than others, we can find a sequence of walls that only miss one barrier somewhere in between. Then we will have a better chance to recognize the almost-enclosed areas and allow the agent to move towards those places. We need to be careful in this case since this can also make our agent be enclosed in a smaller area more often. But this will definitely make the moves more competitive if implemented well. Otherwise, a lot of times the agent is just moving around by itself in places far from the opponent.