COMP 424 Final Project: *Colosseum Survival!*

Minzhe Feng minzhe.feng@mail.mcgill.ca
ID: 260886087
Department of Computer Science
McGill University

Yanis Jallouli yanis.jallouli@mail.mcgill.ca
ID: 260854580
Department of Computer Engineering
McGill University

1. Introduction

For this project, our task is to devise an agent capable of playing the Colosseum Survival game, which involves two players trying to control as many cells as possible on the game board by constantly moving and placing walls around them. The decision making process of this game involves analyzing the board and deciding which position to move to, as well as which direction to put the wall on. In order for a move to be considered legal, the number of steps taken must stay within a predefined upper limit, and the path involved cannot go through the opponent or any existing wall. The game ends when a boundary divides the players into two disjoint areas. Whichever player resides in the larger area wins the game.

2. Program Approach

For our program, we decided to use an alpha-beta pruning algorithm combined with a min-max search tree to determine the best move to be performed. In order to speed up the process, we used a heuristic to estimate how good a position is once a depth of two is reached in the search tree. Our own heuristic involved a linear combination of two different metrics. To further cut down computation complexity, all possible moves in each level are ranked according to another heuristic, and only the first fifteen are explored.

2.1 Theory

A min-max search tree is a game-playing algorithm that assumes optimal game playing by both a player and their opponent. In it, the max-player (us) is attempting to maximize their final score, in this case the number of walls contained in their section. The min-player on the other hand is attempting to minimize our score (thus maximizing their own).

To replicate this algorithmically, we search through a min-max search tree. Here, every possible action of max-player is explored. Then every possible action of the opponent for each of those actions is explored. This continues again for max-player until the game ends. At each node, min-player returns the action that will minimize max-player's final score (return the minimum score of all children). Max-player returns the maximum score - the action that will maximize its score across all children. An example of this is shown below in Figure 1.

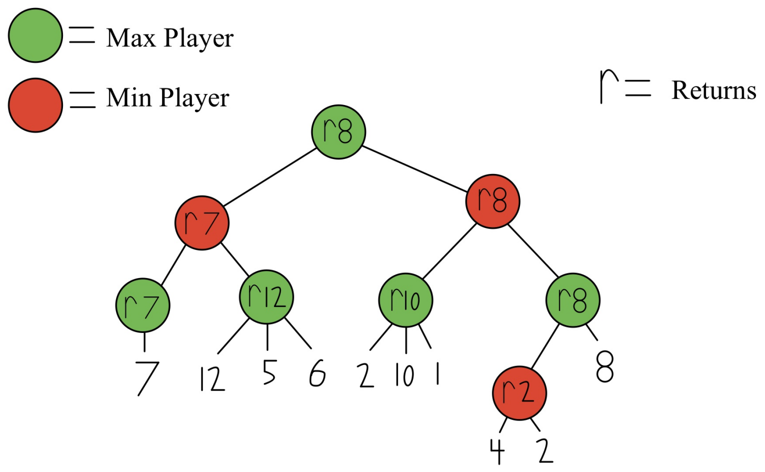

Figure 1: Min-Max game-playing algorithm drawn.

Alpha-beta pruning is used to speed up the exploration of this tree by eliminating unneeded nodes. By remembering the maximum score max-player can achieve (alpha) and the minimum score min-player can achieve (min-player) we can quickly discover if a path is useless for the parent nodes, and thus stop exploring it. For example if min-player can achieve a score of at most 5 with one action, max-player will not choose another path in which it discovers min-player could achieve a score of 1. An example of this pruning is shown in Figure 2.

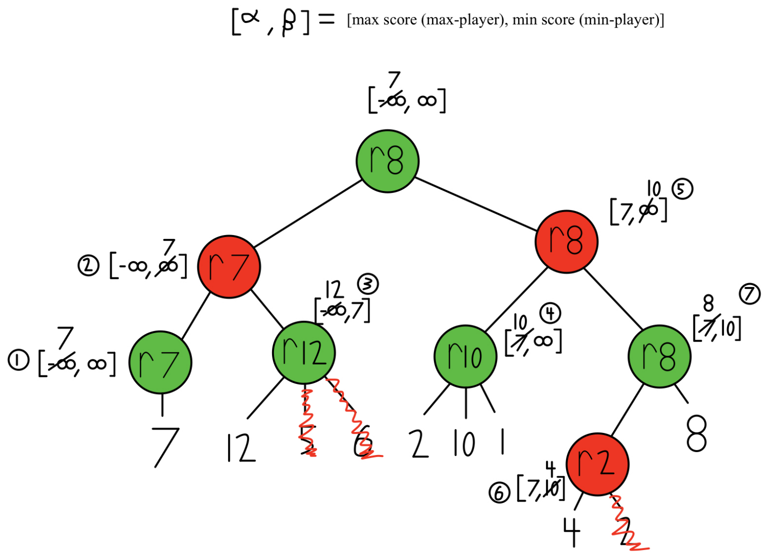

Figure 2: $\alpha - \beta$ algorithm used to eliminate some search paths.

Of course, fully exploring this tree is computationally difficult. As such, we choose to evaluate a heuristic function once a certain depth has been reached as an estimate of how good a state is rather than fully exploring the node. Additionally, only a portion of all possible steps are explored at each node.

2.2 Approach

We implemented alpha-beta pruning as a search. Once a depth of two was reached, a heuristic was evaluated in place of further exploration. Our heuristic sought to minimize the number of immediately sorrounding walls (wall heuristic) and avoid being pushed to a corner (corner heuristic). The two heuristics were combined linearly. The full equation used is shown below. W is the number of walls on the player's tile, while D is the Manhattan distance to the nearest corner of the board. The subscript "pla" is for our value, while "adv" is for the adversary's. "BS" is the edge size of the board.

$$Value = 0.2[0.2 * 0.1(3^{W_{adv}} - 3^{W_{pla}}) + 0.8 * \frac{0.2}{BS}(D_{pla} - D_{adv})]$$

The large number of constants is due to our thorough experimentation of the best weighting to ensure it provided good results without overriding the results of a true endgame analysis.

For the path-finding algorithm, we opted for a standard BFS to first locate all of the possible next steps at each level. Then, we sort them, in ascending order, according to the distances between a possible next step and the adversary's position and pick the first fifteen possibilities to explore. While the heuristics discussed above aim to prevent our agent from getting into adverse situations, this greedy sorting serves to push the opponent into such situations by proactively chasing them down and limiting their choices of possible next steps.

2.3 Motivation

This heuristic was chosen in order to maximize the ability of our agent to push an opponent into a bad location (and stay in a good location) while minimizing the time taken to compute. Computationally, counting the number of walls is extremely easy, as is finding the distance to the nearest corner. A few multiplications later, and an efficient heuristic algorithm plays rather well. Other heuristics were attempted, but did not perform as well (or took too long). The sorting moves algorithm was used to cut down on the needed computations while pushing our agent to proactively chase the opponent, a more offensive game strategy. It is more difficult for an agent to get stuck in a space when it is near to the opponent trying to close it off.

3. Program Analysis

Our final implementation consists of two parts - alpha–beta pruning and path finding. In order for the agent to take a step, it would run an alpha–beta pruning search up to a depth of two, using our path finding algorithm during the process to figure our which steps to explore in the next stage, before outputting the optimal step to take.

3.1 Advantages

The most significant advantage of our implementation is that it's very easy to implement and straight-forward. As mentioned, the heuristics used are quite fast to calculate. The potential steps filtering allows our algorithm to complete within the necessary two seconds with room to spare, even on the 12 by 12 board.

3.2 Disadvantages

The downside of this approach is that estimates had to be made that may not necessarily prove the best approach. For example, the heuristic will prioritize pushing to the middle of the board, even when the middle has become blocked off into an inaccessible portion. Also, our potential steps sorting (to speed up processing) makes it difficult to go around a long barrier to reach an opponent's position, as the agent must first back up to go around. While fast, our algorithm limits the possible moves of the player and cuts off the minmax search quite early (depth of 2) to obtain such speed.

3.3 Failure/Weakness

As mentioned before, our agent fails to account for every scenario. The heuristic means that for much of the game, it is optimizing the score the heuristic assigns rather than the true score an action would lead it to. Playing against the agent ourselves, it was not overtly difficult to cut the agent off while it attempted to optimize the heuristics. Granted however, we did well in making the agent difficult to pin down/trap. The game mentioned is shown below in Figure 3.

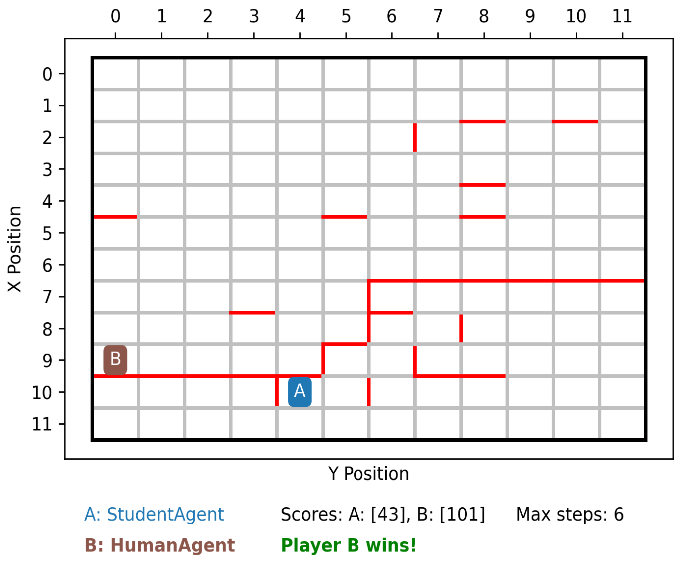

Figure 3: Example of agent losing a game due to following heuristics.

4. Other Approaches

We tried various approaches before finally settling on the implementation described. These mainly consisted of the different heuristics discussed below.

4.1 Random Heuristic

The random heuristic essentially completes the game with random moves, and uses the end result of the game as the estimate for how good a position is. This heuristic performed rather poorly in the end. As seen in games of random agent versus random agent, these games usually end in one agent blocking itself into a small space. Not a good estimate for the amount of space an agent roughly holds.

4.2 Wall Count Heuristic

The second heuristic we developed was integrated into the final algorithm, the wall count heuristic. This measured the number of walls near the player, and used this to estimate how 'trapped' it was. The original function checked a four by four grid around the player and counted the number of walls. In the final implementation, this was adjusted to check the number of walls only on the tile the player was on. This last change was made to improve the speed of the decision-making process.

5. Future Improvements

There are many other heuristics and game-playing strategies that could have been used during this project. To start with, neural networks can be a potentially useful method for playing this game. We decided not to go down this path because it can be very resource-intensive and, due to the "black box" nature of neural networks, finding an optimal architecture can be extremely challenging. Below are two other methods we consider to be viable for this game.

5.1 Monte Carlo

Monte Carlo Tree Search employs a non-random selection method to choose a step to expand upon, according to an evaluation function (e.g., upper confidence trees). It then performs random game-play to improve the estimate of this node's value. One clear advantage for this method is that it keeps the overall computational overhead low (with the random game-play steps), while potentially increasing the reliability of the random simulation by first selecting a promising node.

5.2 Potential Places Heuristic

A few other heuristics may have improved the ability of our agent had we dedicated more time to it. One such heuristic would be to measure the number of available places for the agent and the adversary as an estimate of the amount of space in their "area". This may improve the flexibility of the agent.

6. Conclusion

To sum up, the most important lesson we learned from completing this project is when conventional and straight-forward methods fail to work, how to go deeper and actually analyze the game mechanics to come up with our own optimal heuristic. In the context of this project, we realized running a search all the way to the end of a game is unrealistic, as it's too time-consuming. So, we analyzed the rules and looked for a heuristic to evaluate the board before a game actually ends.