COMP 424 Final Project Report
Game: Colosseum Survival!

Katrina Poulin
ID# 260868280
katrina.poulin@mail.mcgill.ca

Mohamed Mohamed
ID# 260855731
mohamed.mohamed5@mail.mcgill.ca

1. Background

1.1 Game Overview

Colosseum Survival (CS) is a two-player game in which the players need to maximize their owned territory in order to win. Players move around the board and place walls on it until the two players are in separate areas. At this point, the player that owns the largest territory wins.

In Artificial Intelligence (AI) terms, CS can be classified as a *zero-sum* game. In the textbook *Artificial Intelligence: A Modern Approach*, zero-sum games are defined as games in which "what is good for one player is just as bad for the other; there is no *win-win* outcome." [1] Hence, without loss of generality, maximizing player A's score also minimizes player B's score.

We can confirm that this concept applies to CS by examining the game board:

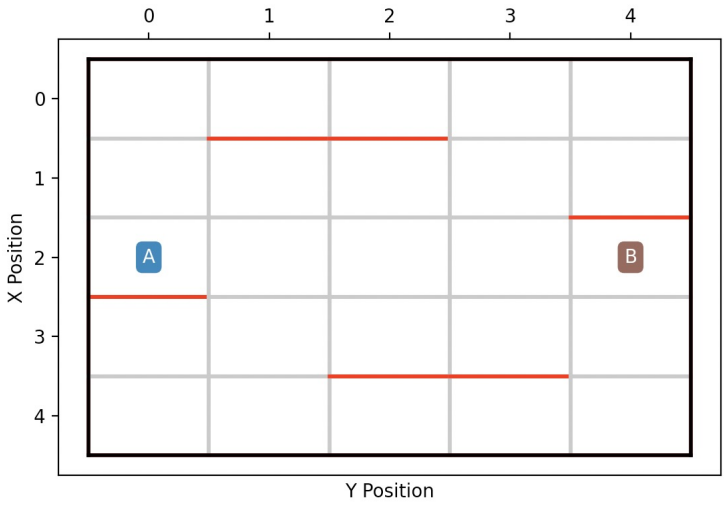

Figure 1: Sample 5x5 Game board for the Colosseum Survival game.

Because both players are on the same board and both need to maximize the number of "owned" squares on the board, the gain of one player results in the loss of the other player. hence, by definition, this game is a zero-sum game.

1.2 Environment

In order to develop an AI agent capable of playing the game rationally, we must determine the information available to the agent and maximize its use. Let's determine some characteristics of the game's environment.

Let us define the developed AI agent to be the *student agent*. In the context of this project, the *sensors* of this agent are the parameters passed as an input to the `step()` function:

- `chess_board`: the state of the game board, providing the location of the walls.
- `my_pos`: the position of the player whose turn it is.
- `adv_pos`: the position of the other player.
- `max_step`: the maximum number of steps that can be taken before placing a wall.

In summary, when calculating the next move, the agent has access to the complete state of the environment. Thus, we say that the environment is **fully observable**.

Since the next state of the environment is always determined by an agent's move and changed only after the agent has played, we say that the environment is **deterministic** and **static**.

Based on the observations made in Section 1.1, we can also state that the environment is a **competitive multiagent** environment. This is due to the nature of most zero sum games as it relies on at least two agents for the game to progress into a terminal state. This environment is competitive as each agent tries to maximize its own performance measure and by definition minimizing the performance measure of the other agent.

We can also identify that the environment is **sequential** since short term action can have long term consequences on the game. This means that our agent has to take into account the future rounds inside the match in order to maximize its own performance measure.

Since the environment has a finite number of distinct states and a discrete set of actions, the environment is considered to be **discrete**, meaning that unless an action is taken by an agent (due to a time out or if a decision is taken), changes in time do not change the game state.

Finally, we can also describe the environment as **known** since all actions taken by the agent have known outcomes.

Our environment is thus *fully observable, competitive, multiagent, deterministic, sequential, static, discrete*, and *known*.

2. Motivation

After analyzing the environment and game type of this project, we narrowed down the algorithms that would be the most efficient in a zero-sum game, while taking reasonable time to implement and not using learning libraries or techniques.

Finding a good move requires an AI agent to generate a set of possible moves and search for the most profitable move to take. Thus, in any case, a function to generate a

tree of possible steps to take needs to be implemented. However, the quality of the agent is mainly determined by the decision-making process, or the search algorithm.

The two search algorithms retained for decision-making were Monte-Carlo Tree Search and Mini-max. Both algorithms have their advantages and their drawbacks.

The Monte-Carlo Tree Search algorithm is relatively easy to implement, since a good part of the random agent functionality can be reused to generate a set of random games. However, since the generated games are not the result of a decision-making process, the outcome of the random games is not fully representative of the agent's performance.

On the other hand, the Mini-max algorithm will be very good at finding the optimal move if all the possible terminal states are known. However, computing the terminal states would result in a very deep state search tree and be very expensive to compute. A solution to this problem is to limit the depth of mini-max to a certain level. Then, the leave states of this level can be assigned a score that aims to predict the score of their future terminal state. Then, the score can be treated as a terminal state score and the agent can make a decision using the mini-max algorithm based on this value. Predicting the score of a state will be achieved using a heuristic described in the following sections. This search algorithm is more representative of the agent's behavior, but is more time-consuming than MCTS. There is a balance to achieve between the depth of the tree and the running time of the decision-making process.

In summary, because of the accuracy of the algorithm and the possible execution time customization, we have chosen to implement Mini-max with a heuristic to determine the score of the leave states. Some testing was performed to determine the ideal depth of the search tree.

3. Program Functionality

Our agent's main functionality is fully indicated in our step function. The program first determines if the current move is our first move, in which case it will initialize the global parameter indicating our current move number. Then, using the parameters given to the step function, it will create a root node with a copy of the parameters given such as the current state of the board, the position of both players and the maximum number of steps. From this root node, and a calculated parameter for desired depth of the tree depending on the move number and size of the board, we will create the tree that will be used for mini-max. We are able to do this given multiple helper functions that help us determine all children nodes from a given node, which are essentially all possible moves which can be computed from the state in the parent node. When constructing the tree, we verify if a given node/state is terminal in which case we will automatically calculate the heuristic for this terminal state being *MyPoints-TheirPoints**1000. Thus, prioritizing this state if it's a win and avoiding it if it's a loss. The algorithm chosen for our agent is mini-max with the heuristic being a linear combination of two parameters indicating the distance to the opponent and the distance of the center, where the distance of the center has more priority.

4. Program Analysis

The choice of our algorithm being Mini-max was based on having a simplistic yet efficient algorithm. We started with Mini-max to test the overall functionality and evaluate if it is worth producing a more advanced algorithm. However, we found out through testing that the main focus of our agent should be the depth of the tree in which case other algorithms were simply unfeasible due to speed restrictions. We optimized the depth of the tree for the mini-max algorithm through the use of multiple hyper-parameters depending on the size of the board and current move number.As the move number increases, our move choices decrease hence allowing us to produce a tree with a bigger depth. For very large board sizes, we also removed some parameters calculated in the heuristic in order to maximize this tree depth. For example, for our algorithm, we found that for a board size of 4, the first 3 moves are computed using a tree of depth 3, the fourth move with a tree of depth 4 and for all other moves a depth of size of 5.

Throughout continuous testing, the heuristic function has been modified significantly. However the distance to the center of the board was always a very good choice of parameter for the heuristic. Another parameter for the heuristic was the number possible of moves of the opponent which we try to reduce as much as possible as to block the opponent. But this parameter was extremely inefficient since it reduced our overall tree-depth by one since its computation is very expensive. We therefore modified this parameter by being the distance to opponent player using level-order graph traversal. The idea behind these two parameters for the heuristic was to prioritize being close to the center of the board while trying to block the opposing player reducing their number of moves. However for the heuristic to be stable no matter the size of the board we needed to have both parameters somewhat consistent. For example if the board size is 6 and the distance to center of the board is 2, it should be somewhat equivalent to a distance to the center of the board of 4 and the board size being 8. To make both parameters consistent the parameter indicating the distance of the board is the ratio of the distance of the board divided by the maximum possible distance to the board. And the parameter determining the distance to the opponent is divided by the maximum possible distance to the opponent. Hence the linear combination of both parameters in this form will be consistent no matter the size of the board.

4.1 Strengths

When testing our program against the random agent, we found that we achieve a high 98% win rate. Ideally this number should be almost 100% since we are playing against an agent that performs no actions based on a clever choice. Regardless of the heuristic, how we found that the most two important factors of the algorithm was its depth for the mini-max algorithm. Therefore, we have optimized our step function to try to maximize our tree depth as much as possible. We found that in latter stages in the game there are less possible moves to compute hence allowing a larger depth for our tree depth in the mini-max algorithm. However, we needed to know how far into the game we are in order to increase our depth. We created a global variable indicating our current move number, which is initialized to 0 with the help of a function that indicates if the current status of the board is our fist move. We used the global variable indicating the current tree depth of the algorithm as a parameter for determining the desired tree depth of the tree for the

mini-max function. Depending on the size of the board, and the current move number, we setup the depth of the tree to be as deep as possible while guaranteeing the computation to not exceed 2 seconds. This optimization is the key to our agent and has multiple optimized hyper-parameters depending on the current move number and the size of the board.

Our second main strength in our agent is the choice of the heuristic. Which is a linear combination of two parameters indicating the distance to the center of the board and the distance to the opponent. With this choice of heuristic, our agent would win 70% of the time against the agent with the heuristic of being a unique parameter determining the distance to the center of the board.

4.2 Weaknesses

The main weakness of our program is it's computation speed. Since we tried to optimize the program as much as possible in terms of maximizing the depth of the mini-max tree depending on the current move number and the size of the board, this has led us to program a high-risk, high-reward agent. The agent could be computing several moves in around 3 seconds, exceeding the maximum computation time, hence penalizing us with multiple random moves throughout the game which might be crucial moves determining a loss. However this choice was made in order to have a massive advantage against other players throughout the last stages of the game where the depth of the tree matters the most.

5. Future Improvements

There are lots of potential improvements that can be made to the algorithm. As we have discussed the most important detail of our agent's algorithm is its tree depth. Therefore we can make optimizations to try to increase the tree depth as much as possible. For example we can instead not consider all possible moves in the beginning of the match since the computation for it is extremely big. We could consider not expanding nodes that are far from the center since they might be pointless. This would save us a lot of computation time and focus our resources of tree depth on moves that are actually important. However, this can be a risky optimization, because if we are playing against a random agent for example, who might not be playing ideally, they might be in the corner in which case for a couple of moves we will not be able to beat them, since these moves are not considered.

Possible improvements would be to improve the parameters chosen for our heuristic function. Both parameters, the distance to the center and the distance to the opponent are made to be values less than 1 since we calculated them to be the ratio in regard the biggest possible value for the distance to the center and the distance to the opponent. More testing can be done to find the ideal linear combination of these parameters.

References

[1] Stuart Russell, Peter Norvig (2021). *Artificial Intelligence: A Modern Approach*. Pearson Education, 4th ed.