COMP 424 Final Project Game: *Colosseum Survival!*

Saagar Arya
260910714
saagar.arya@mail.mcgill.ca

Catherine Van Gheluwe
260485296
catherine.vangheluwe@mail.mcgill.ca

1. Introduction

This report is on our AI agent for a game of *Colosseum Survival*. The purpose of our agent is to verse human, or other AI agents in a game of *Colosseum Survival* and win. Further reading on the project description, and rules to the game are available at the assignment's GitHub repository

2. Technical Approach

2.1 Explanation

Our program uses a greedy approach in-order to play the game Colosseum Survival. The agent starts off by creating a list of all possible moves from the current position of the student agent. In order to find all possible moves, a breadth-first search algorithm is conducted on the current chess board in the radius of max_step around the current position. After getting all possible moves, the agent sorts them into three arrays. One called *higher_scores* which stores all the moves where the score of the agent is higher than that of the adversary, one called *same_scores* and another called *lower_scores* which follow similar logic. The agent uses a modified version of the check_endgame method from world.py to determine the resulting scores of each move on a simulated board.

While the agent is sorting the potential moves, it is also calculating a *heuristic value* for each move. This value is based on the following criteria:

1. Amount of walls in the direct 2 square radius of the agent
2. Distance from the center of the board
3. Amount of walls in the direct 2 square radius of the adversary

The number of walls around the agent is weighted the most, while the distance from the center, and the number of walls around the agent are not weighted as much. We will discuss why we chose these values to calculate the heuristics, as well as the weights below.

Once the agent has sorted the possible moves into the three arrays, and has calculated the associated heuristic value for each move, it goes through the arrays to return the best move.

- It starts off by checking for any moves in the *higher_scores* array where the game ends. If there is any move that matches this criteria, it will return this move as it means that the agent wins. If there is more than one move that matches the criteria, the agent does not care which one it returns as they all result in a win.
- If there were no moves that result in an immediate win, the agent will try to return any other move in *higher_scores*. If there are multiple moves, the agent will choose the one with a lower heuristic value.
- If there were no moves in *higher_scores*, the agent will move on to the *same_scores* array. It will first try to return the move with the lowest heuristic value that does not end the game, as we prefer the game to continue as we have a chance of winning, rather than ending the game at a draw.
- If there are no such moves, the agent will choose a move that results in a game-ending draw.
- If there are no such moves, the agent will choose the move with the lowest heuristic value that results in the agent having a lower score than the adversary that does not end the game.
- If there are no other possible moves, the agent will choose the move that results in them losing the game as a last resort.

Ideally, the student agent will always try to find a move that will maximize its score, and be the best possible score based on our heuristic.

2.2 Motivation for Technical Approach

Given the time restriction in the tournament of 2 seconds per non-initial move, we decided that a greedy algorithm was the best way to go. The theoretical basis of our approach follows general logic of a greedy algorithm where the agent uses its current knowledge of the world around it to make the best decision it can. Our agent looks at every possible move from its current position without thinking about past nor future actions and chooses the move that will maximize its score, minimize the amount of walls around the agent, minimize its distance from the center, and maximize the amount of walls around the adversary. We chose these values as we believe that the amount of walls around the agent is the most important factor in determining the best move, as it will determine whether the agent will be able to move in the next turn, and limiting this limits the chances that the adversary will be able to trap the agent in four walls. The distance from the center is also important as it will determine how close the agent is to the center, and therefore will be in reachable distance of the adversary to cut them off. The amount of walls around the adversary is also important as maximizing this can lead to trapping the adversary in four walls, in which case the agent would win. We chose to weigh the amount of walls around the agent the most, as we believe that it is the most important factor in determining the best move. We chose to weight the distance from the center and the number of walls around the adversary the same, as we believe that they are both equally important in determining the best move. To find all the possible moves around the agent, we use a breadth-first search algorithm.

This algorithm is a common algorithm used in AI to find the shortest path between two points. We use this algorithm to find all the possible moves around the agent, and then use a modified version of the check_endgame method from world.py to determine the resulting scores of each move on a simulated board. To choose which move of equal score to choose, the program uses basic for-loops to cycle through the available moves and finds the one with the lowest heuristic.

2.3 Steps to find this Approach

Our first attempt was a generic greedy algorithm with no heuristic. This agent would simply check to see if the resulting score of a move is greater for the agent than the adversary, and if so, it would choose that move. This agent was not very successful, as it would often choose moves that would result in the agent being trapped in four walls, and therefore would lose the game. This agent had a win rate of 98% against the random agent, however was incredibly easy for a human to beat.

Given the limited intelligence of this agent, we decided to attempt to have the agent find the *next best move* for each possible move, and try to compare those, rather than the immediate moves themselves. This agent had a limited increase in success, but increased the time per move to 9 seconds, which is well beyond the allowed time limit for each move in the tournament. We decided that attempting to find the *next best move* for each possible move was not a viable option, and decided to attempt to use a heuristic to determine the best move.

The first heuristic we used was a simple heuristic that would simply add the amount of walls around the agent. This heuristic version vs the original program would win 60% of the time. In order to improve this, we decided to add the distance from the center to the heuristic, and also add the amount of walls around the adversary. This improved the win rate to 80% of the time. We decided that this was a good heuristic, and decided to use it for the final version of the agent.

3. Pros/cons of Chosen Approach

Pros:

- The agent is able to win 99% of the time against the random agent.
- The agent is able to find the best move in a reasonable amount of time.
- It is not easy to beat this agent manually.

Cons:

- The agent does not learn from past moves, and therefore will not be able to adapt to the adversary's moves.
- The agent does not take into account the future moves of the adversary, so it can be outsmarted.

4. Future Improvements

In the end, our program works very well against the random agent with a win rate of greater than 99% in all runs that we have done. However, since our agent does not look into the future, humans and other agents can likely beat it. An approach that we *did* attempt was to make our agent look at a second step in the future. This technique would involve the agent pretending to take two steps in the future, and then choosing the move that would lead to the best score. Unfortunately this technique increased the amount of time our agent took, with not enough of an improvement in the win rate to justify the increase in time. Another approach we could use to improve our agent in the future would be by using a Monte Carlo approach to look further into the future and make better decisions. Monte Carlo would lead to the agent being able to make better decisions, but would also take longer to run. We could also use a neural network to train our agent to make better decisions, but this would take a very long time to train, and would not be able to be used in the tournament.