COMP 424 Final Project Report

Group Member: Yao Chen (260910483) and Hanzhi Zhang (260908416)

1. Introduction

The objective of this project is to design an AI agent player for the game called *Colosseum Survival!* Which two players move in an $M \times M$ chessboard(M can have a value between 6 and 12) in turns and put barriers around them until they are separated into two closed zones. Each player aims to maximize the number of blocks in its zone to win the game. In each turn, the AI agent will receive the current chess board, the current position of itself and the adversary, and the max available steps. Then the AI needs to choose its next movement and the barrier placement base on the input information.
This report will reflect on how our AI agent was designed and built from scratch. As well as some demonstrations of our algorithms that underlying the AI agent. In particular, this report will also focus on discussion about the advantages and disadvantages of algorithms, and possible improvements that can be done to make the agent behave in a smarter and more efficient way.

2. Motivation and Current Approach

2.1 Motivation

After having a preliminary understanding of the rules, number of trials were tried among team members. Our first instinct is to implement a Monte-Carlo Tree Search(MCTS) or the Minimax algorithm with or without alpha-beta pruning as we assumed they were best suited for game-playing AIs for such chess board games.
Unfortunately, after implementing initial versions of MCTS, we realized that these algorithms for simulating game states added to the complication. Also as the game goes on and the increasing size of the chessboard, the branching factor of game trees grew exponentially in a short period of time. The speed of our program was highly dampened by enumerating millions of possible game states.

2.2 Current Approach

After our initial approaches above, we have two expectations for our program, one is from a software design perspective, we want the code to be well-structured, easy to be implemented and to improve the functionality and debugging. The other is we expect our game agent to have a good performance from a game-playing view.
We simplify the gaming goal for our agent to be ”block the adversary yet keep ourselves alive”. Then our current approach closely resembles the idea of a Heuristic search with some heuristics to compute the cost of action. We use various greedy heuristics to pick the best next step. The primary motivation for using this approach was that it is a simple yet powerful decision-making strategy. Which avoids visiting or simulating a large game tree, while still being able to find a good enough step quickly enough.

3. Theoretical Basis of Our Approach

Overall, as stated in 2.2 Current Approach, a Heuristic search was implemented. By definitions, a Heuristic search is a Best-First search, which is greedy with respect to $f = g + h$ where g is the cost of the path so far and h is a heuristic value to estimate the cost to go, to notice that it turns that computing the accumulated cost would not help with choosing the optimal step as the goal states are dynamic. So the search tree has a uniform cost and is therefore considered to be zero. In general, each time the function `step` called, we will first expand all legal moves from current position `my_pos`. And then for each legal move, we apply the heuristic function to see the 'goodness' of that move.

3.1 Expand

Expand process is implemented under function `find_best_step`, which simply returns all positions that are reachable from current `my_pos` within max step.

3.2 Heuristic function, *h*

After finding all legal moves for this step, the program will pick the move with the best heuristic value. There are two criteria for building this heuristic function. First, in order to gain more blocks, our agent needs to be able to block its adversary, hence needs to get closer to it. We choose to use the Manhattan distancing function to calculate the distance between legal moves and the position of the adversary:

$$\text{Input: p1: position of after a legal move, p2: position of adversary}$$
$$M = |p1x - p2x| + |p1y - p2y|$$

Meanwhile, we notice that it is important to keep a good balance between acting adventurous(always chasing after the adversary no matter what) and acting conservatively(not being blocked by the adversary). So the second criterion is to leave future steps more space if we cannot win in the current step. Also as we move, we hope our agent is able to avoid moving to a position that is already round by barriers, thus we add weight based on the number of barriers at the position. In the heuristic function, use *wall* to denote the number of barriers that surround the position after a legal step. Then for each legal step, its heuristic value can be calculated by the following:

$$h(p) = \begin{cases} M, & \text{if wall} = 0 \\ (M + wall) \times 2, & \text{if wall} = 1 \\ (M + wall) \times 3, & \text{if wall} = 2 \\ 100, & \text{if wall} = 3 \end{cases}$$

Notice that if a block is surrounded by four barriers, then this is not a legal move, so it is no need to consider the situation of $wall = 4$.

3.3 Decision On Barrier Placement

Also, we considered how the barriers should be placed at each step. We follow the assumption that the adversary player is also trying to chase and block us. So our purpose is set to split the space between players in order to increase the moving expense for the adversary agent. Our program will check whether the adversary is adjacent after we move to the next position. If so, a barrier will be placed between two players.

3.4 Decision for Termination

In addition, we also consider the effect of a movement, our program will check if the movement can end the game. When a legal step can end the game with a winning result, the program chooses this step without considering any weight. On the contrary, a huge weight will be assigned to a move that loses the game.

4. Advantages and disadvantages of our approach

4.1 Advantages

In general, our program is built by light weighted and fast decision-making model, with $O(n+d)$ complexity. In each step, a simple tree with a depth of one will be built to find the best position to move to. Then the direction of a barrier will be decided to lower the utility of the adversary(maximizing its moving expense). And is able to end the game when the winning is obvious. On average, it took 0.5 seconds to complete one game, which is efficient enough.

4.2 Disadvantages

The core disadvantage is that the underlying logic of our program is the strategy of live fast dies fast. Instead of considering all steps till the end of the game, our program only considers the optimal current step without making predictions on the adversary ahead. This leads our agent to a short-term local optimal. However, when playing with an agent that has the ability to take steps of prediction, our program is obviously not smart enough.
Another drawback of our AI is the ability to detect and maximize its own zone is weak. A piece of obvious evidence is that our AI is not able to identify a continuous wall, which would easily ahead to a dead end. As shown in Figure 1, the area where the red path pointing to belongs to player A implicitly, any human player won't go that way. However, that is what our agent chooses to do. The crux is the use of Manhattan distancing, we ignored the existence of the continuous wall. And focus on the primary task is to get close to the adversary. And we are not able to implement this detection due to the lack of time. Other than that, there are existing more hidden tricks that are not known by our agent(not reflected by our heuristic function). In addition, this assumed an opponent using a similar strategy to us. So it is not able to beat a human player when the human player only pays mild attention.

5. Other approaches

We had other two approaches. As we state in our Motivation, before the current version, we had considered the large branching factor that each step has, so it seems like Monte Carlo Search is a good possible approach. However, our implementation exceeds the run time limitation with poor performance.

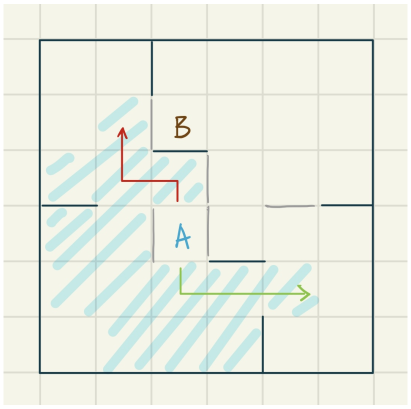

Figure 1: A as our agent, B as its adversary. In our program, A will choose to go to the red path, whereas the green path is actually a better decision

The second approach is an optimization version of our current approach, base on our current agent, we implemented a version with a 2-step forward check. Which will run two steps ahead to check how likely a step be an optimal step. And it is able to win 80% of the game with our current version, and also able to let a human player feel mild challenging. However, our likely computation exceeds the run-time limitation, and we haven't got time to optimize it.

6. Improving our player

Since our average run time is only 0.5 seconds per game, and the limitation is 2 seconds. Clearly, there is a lot of optimization that we could make to improve our agent. By the disadvantages stated above, to solve the problem of dead-end, one way of improvement is to let our agent able to detect the correct path to block the adversary by some algorithm such as Dijkstra's algorithm. The use shortest path as a sub-heuristic could help our agent to correct direction.

We consciously decided not to simulate various game states after our try of MCTS. In hindsight, if the heuristic about the shortest path is implemented, the size of state space would be reduced. Knowing that our two-step forward checking model has an 80% winning rate against our current model. Therefore it would have been a great advantage to build prediction trees when we can confine our steps in a certain area rather than all possible areas. So running a larger search tree could probably add more intelligence to our agent.

Other than improving our current program itself, recall that we simplify the goal of our agent to be live fast die fast. Yet the actual goal of this game is to maximize the utility(size of a closed zone). We could implement our model from the perspective of gaining more blocks. This could be implemented by algorithms such as MCTS with the Minimax algorithm.