COMP 424 Final Project Game: *Colosseum Survival!*

Authors: Karim Elgammal and Anaëlle Drai Laguéns

1. Introduction

The main task of this project is to attempt to optimize an AI agent to play ***Colosseum Survival!***, a 2-player board game. The game starts with a $M \cdot M$ chessboard with randomly positioned barriers. The two players take turns moving vertically or horizontally through the grid, respecting the limit of the maximum step number of $K = (M+1)//2$, and they position a barrier in one of four positions: up, down, right, or left.

The players are not allowed to move diagonally, cross barriers, pass or take the position of their adversary, or put a barrier in a direction where a barrier is already present. The game ends when the two players are separated by the barriers in two bounded regions and the score of each player is the number of blocks in its region.

Our goal for this project was to build an agent with the maximal winning rate, able to win against a random player, human players and even other AI agents.

2. Motivation

Before starting the implementation of our agent, we decided to analyze the algorithms seen in class to determine what algorithm would prove to be the most adapted to this problem. Because the branching factor b of ***Colosseum Survival!*** is high considering there are M^2 cells in the board and 4 barrier choices per cell, and because the solution depths m can also be high, we decided to immediately eliminate MiniMax search which has a worst-case time complexity of $O(b^m)$ [1]. Even with the addition of alpha-beta pruning, the algorithm is really expensive and we have a limited time to perform each move. Moreover, MiniMax is optimal if both players are playing optimally according to a certain evaluation function, and our agent will not be faced with optimal players (human, random or even other student agents are most likely not optimal), which will additionally not be using the same evaluation function (because there is no obvious precise strategy in this game) [2].

We thus decided to focus on using the Monte Carlo Tree Search algorithm, which seemed much more adapted to this problem as it does not depend on a precise evaluation function, but rather on several random simulations of the game. Our strategy was to start by implementing an MCTS algorithm, analyze the results and then iterate based on the observed outcome. However, our implementation of MCTS was time-consuming and the results were almost not greater than 55% against a random agent, which was actually not far from the results obtained when a random agent played against another random agent. Even if this observed outcome might have been due to problems with our implementation, we decided to change our approach and consider other options because our time-consuming MCTS implementation did not show satisfying behavior.

We started playing against the random agent as humans and analyzed the strategies that emerged: we almost always focused on chasing the opponent and trying to circle him with barriers. That's when we decided to try a different algorithm, a "chasing" algorithm which would reflect our human approach to solving this problem, analyze the results and iterate again. Our objective with the chase algorithm was to maximize the number of barriers surrounding our opponent and thus try to block them in a bounded region with the smallest amount of blocks, resulting in a win for our player. This approach resulted in a high percentage win (around 75%) against random agents, we thus decided to focus on this chasing algorithm, and tried to optimize it as much as we could. We reached a winning percentage of almost 99% when we also decided to try minimizing the number of barriers surrounding our agent and thus minimizing the risk of being blocked in a small bounded region, resulting again in a more likely win for our player.

3. Theoretical Basis

3.1 MCTS

The Monte Carlo Tree Search algorithm is an application of the Monte Carlo method which consists of using repeated random sampling to estimate some numerical quantity [3]. For game playing, it consists of generating random game simulations for each possible move and selecting the most promising ones, which have a higher winning ratio. MCTS consists of four different steps. First, the selection process which uses a tree policy on nodes seen before to select a promising path in the search tree. The policy we used is an Upper Confidence Trees policy, which balances both exploitation and exploration. This means that in addition to prioritizing promising nodes, it also explores nodes that have not been explored a lot. The next step is the expansion step, where we expand the tree when we reach the frontier (leaf nodes). Then the simulation, which consists of performing sample roll-outs using a default policy from an expansion node, and recording the outcome of each randomly generated game. Finally, back-propagation, where we update the value and visit counts for the states visited during selection and expansion after the roll-outs.

3.2 Chase algorithm

The implemented chase algorithm relies on two greedy depth-limited heuristic search algorithms. The algorithms estimate the node successors' distance to the goal and expand the successor node with the shortest distance according to the heuristic. If the maximum depth is reached and the goal is not found, it backtracks and searches for the next node with the shortest distance. The heuristic used to estimate the distance is the distance based on a relaxed version of the game, it is thus an admissible heuristic, which means it is optimistic compared to the real distance to the goal, and always less than or equal to the actual distance [4].

4. Algorithm description

4.1 First Iteration

The first step towards the implementation of our algorithm was to implement a check to determine whether surrounding the opponent was possible (i.e., an adjacent cell was reachable using at most max_step, the maximum number of steps). To implement this check, we defined a recursive method `self.reachable()`. It takes as arguments the current position of the player, the position of the cell he wants to reach, the opponent's position, the current chessboard state, the maximum step, and the current step (to implement the recursion). This method corresponds to a greedy depth-limited DFS, where the function first determines what steps the current player can take at the current position (x,y) by applying all the possible operators (i.e., up, left, down and right which correspond respectively to (x-1, y), (x, y-1), (x+1, y) and (x, y+1)).

The `self.reachable()` method then filters that list and removes all the invalid generated positions. The performed checks are the following:

- The next position does not go out of the bounds of the chessboard.
- The next position is not the opponent's position.
- There are no barriers between the current position and the next position.

To increase the performance and make the algorithm more efficient, we sort the generated list according to a heuristic function that computes the distance to the position we want to reach (by simply summing the x and the y difference between the two positions). This allows to explore cells that are more promising according to a heuristic that evaluates the distance based on a relaxed version of the game (it considers there are no barriers on the way and no adversary on the board) [5].

The method then performs a recursive call to itself on the next position with the closest predicted distance, increasing the current step by 1. It returns true if the objective position is reached, and false if the current step is equal to the maximum step (i.e. we have reached the maximum step number) or we have explored all the valid successors (i.e. next positions) and not one of them reaches the desired position.

After implementing that method, we moved on to implement `self.step()`, the method that would return the new position of our agent. The main goal of this method was to find a new position adjacent to the opponent and place a barrier toward him (if a barrier was not already there). After checking for all adjacent positions if that was possible using the `self.reachable()` method we just described, if no position was found, we decided to simply return a random position, by calling the `self.random_step()` function, copied from random_agent.py and slightly modified to make it functional in this class as well. The success rate against a random agent of this first implementation of our agent was around 75%.

4.2 Second Iteration

We decided that a winning rate of 75% was not enough, and decided to work on the biggest and most obvious possible improvement: finding a better solution than generating a random position when the cells adjacent to the opponent were not reachable. The strategy we chose was similar to our adopted behavior as humans when we played the game, we decided to modify the algorithm in order to get as close as possible to the opponent instead of performing a random move, in order to possibly reach and block him in future rounds.

We implemented a second recursive function `self.find_closest()`, which was an algorithm similar to the one previously described in `self.reachable()`. The same logic of a greedy depth limited DFS is applied, with a heuristic function and sorting of the successors so that they are chosen in order of increasing distance (i.e. closest first). The main difference is that this algorithm is greedy and not optimal, as it considers that the first valid solution found is the best one and returns it. This new method allowed us to return a good-enough position, closer to the opponent, whenever he was not reachable. The success rate was increased to around 82% against a random agent with this new strategy.

4.3 Third Iteration

Even though our agent performed well against a random agent, after watching it play against humans, we took note of several sub-optimal behaviors, often leading to loss against human players, that could easily be avoided and we decided to add additional checks to our implemented `self.step()` algorithm, to increase the robustness and winning rate of our chasing algorithm.

The first check we added was to implement a `self.win_check()` method by taking advantage of the `check_endgame()` method from world.py. This method checked whether putting a barrier at that position and direction caused the game to end, and if it did, if it resulted in an opponent win. This method was added as a check whenever the agent had to return a position (either for cells adjacent to the opponent or in `self.find_closest()`), to make sure the return position did not imply a win for the opponent and ruling out all the lost games caused by our agent placing a barrier that would result in an opponent win.

The second check we added was a safety check that confirmed we did not put the agent in a dangerous position when finding the closest position or when moving to a cell adjacent to the opponent. The method `self.block_check()` determined whether the cell we were moving to was surrounded by 2 or more barriers. If the cell had 3 barriers, then moving to the cell and adding a barrier would immediately mean losing because our agent would circle himself. If the cell had 2 barriers, then moving to the cell and adding a barrier would mean the agent would be surrounded by 3 barriers, thus making it easy for the opponent to simply add a final barrier, resulting in a lost game for our agent. The method performing the check is called self.block_check() and it simply checks that the amount of barrier at the cell we're moving to is greater or equal to 2, it also checks for 1 and 0 barriers whenever the agent is at the chess board limit.

The success rate against a random agent immediately increased to 95% with those additional safety checks, and it made it hard for humans to win against our agent. However, a new problem appeared. Our `self.find_closest()` method would sometimes find no cell satisfying those conditions and return nothing, which is a situation we had not thought of handling before, because we thought the state space was large enough, and there would always be at least some states satisfying the conditions. In the rare cases where no position was found, the agent simply performed a random step using `self.random_step()`.

The final winning rate for our agent after these improvements was an average of 98% against a random agent, and an estimated 50% when confronted with humans. Testing it against humans involved running a game against our agent 20 times; alternating the human player between both participants of the project. Humans could not find or think of any fixed strategy to counter its moves, which we determined was sufficient proof that our agent was robust enough. Further, it gave us incentive to believe that our agent was in fact acting rationally.

5. Possible future improvements

The obvious and easily implementable improvements for our agent mostly consist of fixing its current problems and weaknesses. The main problem is the choice of a random step when the method that finds the closest position to the adversary does not return anything. Improving the `self.find_closest()` function which is currently neither optimal nor complete would be the best way to strengthen our chase algorithm. The second possible improvement is extending checks for edge cases, a concrete example would be avoiding positions that are at the boundaries of the chessboard and positions further away from the center because it is easier to block our agent there.

The larger-scope possible improvements are to implement a more complex and complete strategy for our agent. Because our run time is exponentially faster than the allowed limit for each move, we could have implemented more complex and higher-quality heuristics and algorithms. For example, combining the strength of our chasing strategy with the results of an MCTS algorithm or even with machine learning, using results from previous games to choose future moves more wisely could increase the win rate of our agent to 100% against a random agent and a better score than 50% against humans. A further improvement which may be deemed outside the scope of our investigation and may have yielded good results, could be shifting the axioms of search from the atomic states and atomic actions to a planning as search problem. That is; designing a set of states and goals as conjunctions of predicates along with operators which are defined with a set of preconditions and post-conditions about the nature of the game.

6. Advantages and disadvantages

The greatest advantage of our agent, is that it is fast and it combines an offensive and a defensive approach for ***Colosseum Survival!***. The main objective of our agent is to chase the opponent and get closer to him in order to block him. However, it will almost never perform a move that puts it in a dangerous situation by always checking for the risk of losing before performing any step. This is an advantage compared to any other agents, that did not choose a greedy approach, optimizing moves in the long run but without checking for immediate danger. Our agent is efficient and it immediately performs offensive moves, meaning it is more likely to win against such agents if it manages to block them quickly, because a complex agent optimizing moves in the long run will perform moves that are apparently random at the beginning (because of the wide range of possibilities), however, as this range narrows, the likeliness of our agent winning diminishes over time.

The main disadvantage of our agent is that it relies only on one strategy. Even though we focused on implementing the best possible algorithm for an agent that relies mostly on chasing the opponent, if an agent has an algorithm that chooses to move in a way that counters our blocking strategy properly by performing safety checks, then our agent is not efficient at all. Agents implemented with a strategy that checks for more complex strategies, and choose safe moves while maximizing the long-run chance of winning are also most likely to win against our agent.

7. Conclusion

The final implemented agent relies on a strategy that tries to mimic human strategy by combining an offensive and a defensive approach. It tries to chase the opponent and circle him with barriers while avoiding possible dangerous positions. The overall winning rate of 98% against a random agent is satisfactory, and the winning rate of 50% against human players is a sign of the robustness of our algorithm, which was developed using heuristics and relies mostly on two greedy depth-limited DFS algorithms. The agent is also consequently efficient and fast, leaving room for considerable improvement and the possibility of reaching a 100% winning rate against random agents if it is combined with other more complex strategies and algorithms.

8. References

[1] David Meger, COMP 424 - Artificial Intelligence, Topic: ”Searching Under Uncertainty”, slide 97, McGill University, Montréal, Fall 2022.

[2] David Meger, COMP 424 - Artificial Intelligence, Topic: ”Game Playing Continued”, slide 48, McGill University, Montréal, Fall 2022.

[3] David Meger, COMP 424 - Artificial Intelligence, Topic: ”Game Playing Continued”, slide 57, McGill University, Montréal, Fall 2022.

[4] David Meger, COMP 424 - Artificial Intelligence, Topic: ”Game Playing Continued”, slide 45, McGill University, Montréal, Fall 2022.

[3] David Meger, COMP 424 - Artificial Intelligence, Topic: ”Informed Search and Local Search”, slide 32, McGill University, Montréal, Fall 2022.