COMP 424 Final Project Game: *Colosseum Survival!*

Course Instructor: Jackie Cheung and Bogdan Mazoure
Authors: Ian Dexter (260839862) and Alex Bertrand (260861032)
Due Date: April 12th, 2022, 11:59PM EST

1. Introduction

The goal of this project is to develop an agent to play the game ***Colosseum Survival!*** against both random opponents and other student submissions. Our measure of success in achieving this goal is the win rate of our agent against these adversaries. A win is defined as a board configuration in which no path exists between our agent and the opponent, and our agent has control of at least 55% of the entire board. In addition to the rules of the game, the agent is subject to both time and space constraints.

2. Choice of algorithm

Two approaches were initially considered for this project: Minimax search with Alpha/Beta pruning and Monte Carlo Search Trees (MCST). Although we hypothesized that a good implementation of either technique could reliably defeat both random and intelligent adversaries, we identified several characteristics of ***Colosseum Survival!*** that seemed more suited to the latter technique. Firstly, Mimimax search is only guaranteed to be optimal against a rational opponent. As our algorithm is mostly evaluated against random agents, this advantage is frequently irrelevant. This argument extends to a Minimax algorithm with a heuristic function. Optimality is not guaranteed in games against other student submissions either, as each agent would most likely use a different heuristic. Another disadvantage of Minimax is the considerably high branching factor of ***Colosseum Survival!***, especially at higher board and step sizes. As time and space constraints are significant in this project, we hypothesized that the searchable space in a Minimax game tree would be too shallow to provide good results. An excellent evaluation function would therefore be essential, especially for larger boards where terminal states only occur after many moves. However, neither author was confident enough with ***Colosseum Survival!*** strategy to hand-craft such a function.

Much of the framework for a basic MCTS algorithm was provided either in-lecture or as a part of this project. Many elements of the random ***Colosseum Survival!*** agent could be re-purposed to perform MCST random simulations during the default policy phase. Furthermore, the UCT tree policy is simple to implement, robust, and effectively balances exploration and exploitation [3].

3. MCST Design Choices

3.1 Tree policy: UCT

As outlined in lecture, the UCT function used to select a leaf node is:

$$Q(s,a) + c \cdot \sqrt{\frac{\log n(s)}{n(s,a)}}$$

where c is a tuned hyper-parameter. Based on qualitative observations of game play on a constant size board, the optimal value is approximately $c = \sqrt{2.2}$, although promising results were observed as low as $c = \sqrt{2}$ and as high as $c = \sqrt{5.5}$. Above this, the moves became extremely scattered such as in Figure 1.

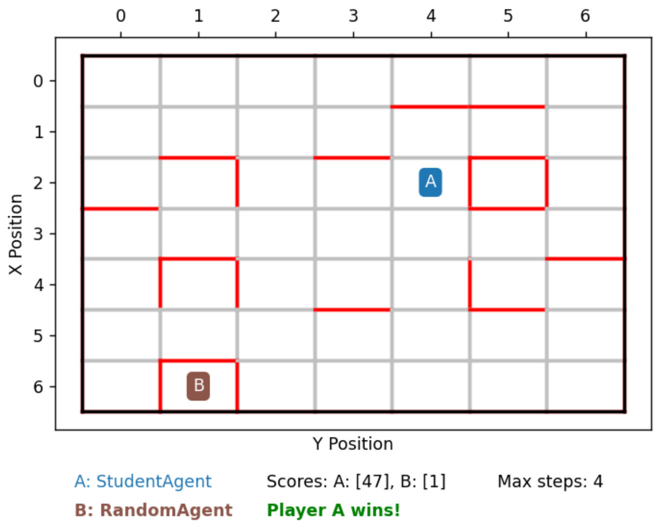


Figure 1: The result of a game with $c = \sqrt{6.5}$. The agent favors exploration too heavily.

3.2 Default policy: Random simulations

Once a leaf node had been selected by the UCT tree policy, random simulations predicted the outcome of a game from that state. These simulations consisted of repeatedly passing the board state and agent positions to the `find_moves()` function, which returned all the possible moves for the agent whose turn it was. One of these moves was randomly selected until a terminal state was detected by an adapted version of `check_endgame()`, which also served as our evaluation function. The output of this function is a value of $+1$ when our agent wins, and 0 otherwise. This value was updated at the selected leaf node and back propagated to all of its parents. An alternative implementation for `check_endgame()` consists of returning the number of squares that our agent controls at the end of the game. However, the primary objective of the agent is to win games frequently, regardless of the quality of each win. We suspected that choosing such a function would discourage the agent from seizing clear opportunities to win in cases where the adversary controlled a significant portion of the board, such as in Figure 2 b.

3.3 Timer

As a game of ***Colosseum Survival!*** progresses and the number of walls increases, the movement of each agent is further restricted. As a result, the branching factor of the game tends to decrease. By running random simulations in a time-based loop, we hoped that more simulations could be run per turn as the game developed. This would be especially useful in small boards where the branching factor decreases significantly with each wall placed, and the likelihood of getting completely surrounded by walls increases.

The extended time limit for the first move was allocated to running more simulations. We considered using this time to plan a strategy over several moves, but figured this would seldom be effective given the unpredictability of most opponents.

3.4 RAVE

For this particular use-case, we decided against using a technique such as RAVE to share information between states in the search tree. Here, what makes a move "good" appears very context-dependent: a move that in one board state could trap an opponent would in most states do basically nothing, and could even result in a loss. This is different from games like chess, where certain moves (e.g., castling or advancing center pawns) are generally strong regardless of the exact board state. Therefore, we hypothesized that the resources necessary to support RAVE should rather be allocated to perform more random simulations.

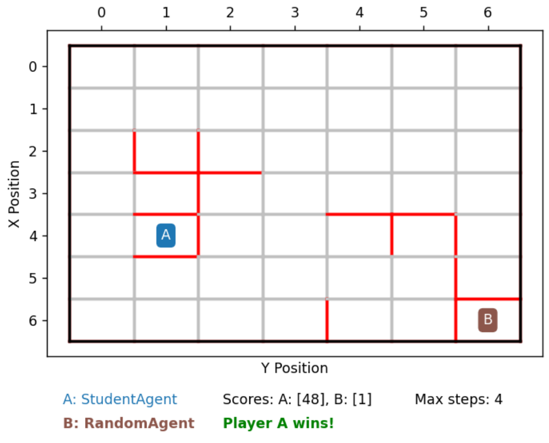
4. Naive MCST Observations

Our first version of MCST simply combined a UCT tree policy (with a tuned hyperparamater c) with random simulations. Although this agent reliably beat a random counterpart, this did not appear to be the result of any particular strategy. Our success seemingly depended on the random agent eventually making a series of bad moves. In addition, our agent often placed walls in a grid-like or web-like pattern which a smarter opponent could easily exploit to trap us (Figure 2 a). We therefore attempted to design a heuristic function, the output of which would be used in conjunction with UCT values of nodes to make more proactive moves. For example, (Figure 2 b) depicts our MCTS agent trapping its adversary in a losing position. Despite the frequent aimlessness of the naive MCTS implementation, it was observed that this algorithm would reliably place the final block to win a game if the adversary had no valid moves in 3 directions. Since the actions chosen by MCTS are heavily influenced by the result of simulations against an adversary that often self-destructs, it is possible that the UCT values of aggressive moves are only significantly higher if victory is immediate.

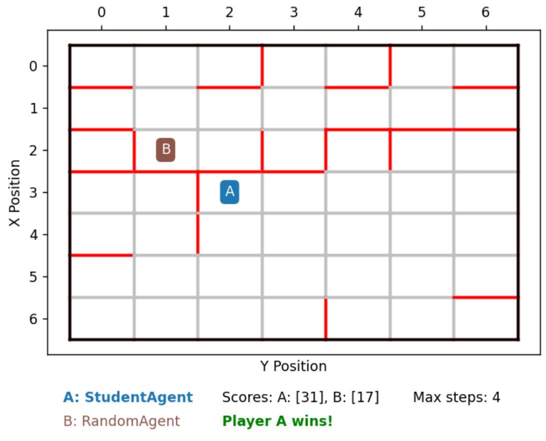
5. Heuristic Functions

5.1 Open Space

The first problem we set out to solve using a heuristic was the agent's tendency to stay in one area and stall until the opponent randomly played a losing move. Since the area around our agent became filled with walls, one or two well-placed walls from a better-than-random opponent could isolate our agent. To counter this, we developed a heuristic that counts the



(a) Parameter $c = \sqrt{6}$



(b) Parameter $c = \sqrt{5.5}$

Figure 2: The result of two games with a naive implementation of MCTS. Our objective in designing a heuristic function was to make our agent (blue) actively win games (b) as opposed to delaying until the opponent (brown) made a mistake (a).

average number of walls within a square of a given radius around the player. This number was normalized and applied as a penalty when deciding on the move to take. Testing with different values for the radius indicated that a radius of 1 (i.e., a 3 by 3 square centered on the agent) offered the best performance.

In general, this heuristic also incentivizes the agent to move into areas with fewer walls. This is convenient, as large areas with few walls will tend to be part of the winning side when endgame is reached. It therefore serves to both keep the agent mobile and to help it find potentially good board positions to control.

5.2 Centrality

We also found that, particularly in the early stages of the game, our agent often stayed towards the perimeter of the board. At first, we gave the agent the benefit of the doubt that it had found a good strategy. However, it soon became clear that this was a poor tactic, as the agent frequently trapped itself in a corner and did not leave enough escape routes to more open areas. At the center of the board, the agent can move to a wide variety of locations throughout the arena, rather than confining itself to one corner (at least in the early game, before many walls have been placed).

To encourage the agent to control the center of the board, we added a penalty to moves proportional to their distance to the center; the penalty is maximized at the corners of the board and minimized at the center. Like the previous heuristic, the value was normalized to ensure the size of board didn't affect the importance of the heuristic, and to allow for easier weight tuning.

5.3 Aggression

The previous two heuristics provide a good default strategy for our agent to follow early on, by rushing the center and attempting to find a wide-open space within the board to play from. However, following this tactic is not always the best way to play. We noticed that our agent sometimes ignored sequences of strong moves against poorly-positioned opponents, and instead chose to continue playing around the center.

We wanted our agent to aggressively capitalize against opponents that managed to get into disadvantageous positions. We defined "disadvantageous" using the Open Space heuristic described earlier, except centered on the opponent rather than our agent; adversaries with many walls around them had fewer movement options and could be more easily isolated in a small zone. To encourage our agent to pursue in these situations, we gave a bonus to moves near the opponent scaled by "how bad" the opponent's position was (the value of the opponent's Open Space heuristic).

5.4 Heuristic Weights

The goals of the "positioning" heuristics (Open Space and Centrality) can be opposed to the goal of the Aggression heuristic. The former attempts to gain and maintain a good position early on, while the latter attempts to close out the game after letting it progress for a while, which often involves moving away from the center and into heavily-walled areas. To incorporate this into our agent, we geometrically reduced the *positioning weight/aggression weight* ratio, thereby emphasizing aggression as the game proceeds.

We tuned the heuristic weights manually, by observing our agent play against both the random agent and different versions of our own agent.

5.5 Heuristic Decay

The heuristics were mainly designed to give our agent a simple but effective strategy towards the beginning of the game, when the branching factor is too high to search to a significant depth. As more walls are placed, the branching factor decreases enough for the MCTS to provide a more accurate understanding of the game state. It is therefore less important, and sometimes even counterproductive, to act according to the heuristics. For this reason, we decided to make the agent's reliance on the heuristic decay geometrically by multiplying its weight by a fractional constant each turn.

6. Final version

Figure 3 shows an example of the moves taken by the final version of the our agent against a random opponent. Unlike most runs with the naive MCTS implementation (Figure 2 a), this agent clearly engages and confines the adversary. This game ends with our agent placing a wall at (6,5,left), effectively trapping the opponent in a 2-by-1 rectangle.

7. Strengths and weaknesses

Based on our testing, the combination of Monte Carlo simulations and heuristics seems to provide a good balance of cohesive early-game strategy from the heuristics and effective late-

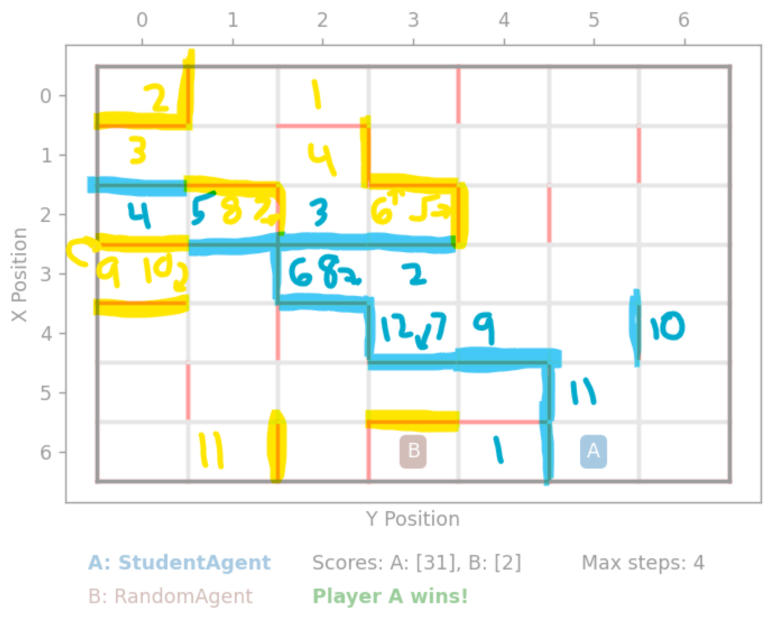


Figure 3: The result of a game with $c = \sqrt{2.2}$ and tuned heuristics. The numbers correspond to the sequence of movies for each agent and the highlighted walls are those placed at each move. The initial state of each agent is denoted 1. Note that the student agent (blue) moves first and the path of the random adversary is in yellow.

game play from the MCST. The agent can achieve near-perfect win rates against random opponents, and appears to act intelligently against more advanced adversaries.

One significant weakness of our design is the way we tuned the heuristics. Though we tried to be as objective as possible, the hand-tuned weights were ultimately chosen from what we intuitively felt the agent should have done differently based on a small sample of games. The weight decay also isn't affected by the board size; this means that the agent transitions from its early-game tactics to its late-game tactics in as many turns on a 12 x 12 board as it does on a 6 x 6 board, despite 12 x 12 games usually lasting much longer.

We also developed our agent by playing against either random opponents or different revisions of itself. This means that our opponents were either very bad (the random agent) or followed a familiar strategy. High-performing adversaries could employ new strategies that might nullify our aggressive game play, or worse yet learn to capitalize on our heuristics. For example, a human agent could quickly learn to encircle our agent in the centre of board. Finally, our agent does not distinguish between wins and ties in which it controls over 50% of the board. Therefore, it may be over-eager to accept ties when a more convincing winning move exists. However, this also implies that it looks to tie when in a losing position instead of searching for overly ambitious winning moves.

8. Future Work

To address the arbitrariness of the heuristic weights we could use a more exact procedure, such as a genetic or hill-climbing algorithm, to tune them. This would also allow for much larger game sample sizes, as it wouldn't have to be done manually. The heuristic decay rates can also be investigated to see if decaying at different speeds based on board size improves the agent.

Another way to significantly improve our algorithm would be to make it more computationally efficient. For example, our current implementation checks if the game is finished after every move, which takes up a significant portion of the allotted calculation time. There are some optimizations that can be used to reduce the number of checks that need to be done, especially early in the game. For example, a check only needs to be done if the move places a wall between two other walls; otherwise, it's impossible for that move to disconnect two areas.

It may also be possible to devise ways of rejecting bad moves before exploring them, an idea similar to Alpha/Beta pruning. While developing our agent we attempted to incorporate our heuristics into the UCT calculation, but the results were unsatisfactory. We still believe there is potential for this idea, it's just a matter of finding the best way of implementing it.

9. Credits

- Lecture slides: L9-Monte Carlo Tree Search, L8-Games
- `random_agent.py` and `world.py` implementations
- Article on MCTS for tic-tac-toe in Java [2]
- Blog post on MCTS for AlphaGo [1]

References

[1] Monte carlo tree search tutorial: Deepmind alphago, Apr 2020. URL `https://www.analyticsvidhya.com/blog/2019/01/monte-carlo-tree-search-introduction-algorithm-deepmind-alphago/`.

[2] Oct 2020. URL `https://www.baeldung.com/java-monte-carlo-tree-search`.

[3] Maciej Świechowski, Konrad Godlewski, Bartosz Sawicki, and Jacek Mańdziuk. Monte carlo tree search: A review of recent modifications and applications, 2021. URL `https://arxiv.org/abs/2103.04931`.