Colosseum Survival Report

Zu, Greta
greta.zu@mail.mcgill.ca

Xiao, Quinn
quinn.xiao@mail.mcgill.ca

December 6, 2023

1 Introduction

The main motivation behind the coding of our agent is writing an algorithm that is capable of navigating through the dynamic environment of the game *Colosseum Survival*. Our ultimate objective is to win against sub-par opponents. Thus, our student agent must not only explore possible moves efficiently but also incorporate strategic considerations. In tackling this challenge, we used some of the problem-solving methods studied through this course: A* search, Breadth First Search (BFS), simulations, and heuristics.

Our approach integrates the strengths of A* search and Breadth-First Search (BFS) to explore feasible moves and identify the most promising paths based on a combination of distance metrics and heuristics. The inclusion of heuristics, such as offensive and defensive considerations, enhances the agent's decision-making by prioritizing moves that contribute to offensive strategies, while also accounting for defensive maneuvers in unfavorable positions. The combination of these two algorithms strategically addresses the dual challenges of short-term tactics and long-term strategic planning.

Additionally, our strategy extends beyond exploration. Inspired by the simulation step in Monte Carlo Tree Search (MCTS), our approach involves simulating potential moves and dynamically adjusting heuristics based on the number of moves left. As a result, our student agent is able to anticipate the consequences of wall placements and simulate game progression. This nuanced decision-making, balancing offensive and defensive priorities, defines our agent's strength.

Our agent's strength lies in the synergy of A* search, BFS, simulations, and heuristic refinement. This strategic emphasis on offensive and defensive considerations positions our agent as a competitive player against `random_player` and inexperienced players in *Colosseum Survival*.

2 Agent Design

2.1 Data Structure Overview

Before delving into the specifics of our exploration strategy, it's essential to understand the underlying data structure that plays a crucial role in our agent's decision-making process. We use a structured list of sublists to keep track of moves and their associated heuristic information. This data structure allows for efficient organization, retrieval, and analysis of potential moves during the exploration phase. Each sublist in the data structure contains the following components in order:

1. Priority (p): A numerical value representing the priority of the move. This priority is determined based on a combination of heuristics, offensive and defensive considerations, and other strategic factors.

2. Score (s): The score assigned to the move, reflecting its desirability. The score is updated during simulations and contributes to the overall evaluation of potential moves.

3. Number of simulations (n): The number of simulations associated with the move. This information is crucial for assessing the average score from all of its simulations.

4. Position ((x, y)): The position on the game board where the move is to be executed. This allows for easy identification and visualization of the chosen moves.

5. Direction (dir): The direction in which the wall is to be placed. This component specifies the intended action associated with the move.

This structured data organization facilitates efficient retrieval and interpretation of relevant information during the decision-making process, contributing to the overall effectiveness of our agent's strategy. It serves as the foundation for both A* search and Breadth-First Search (BFS), as well as the subsequent simulation and heuristic adjustment steps in our overall strategy. The following sections will elaborate on how these components interact and contribute to the agent's decision-making process.

2.2 Timeout Mechanism

Crucially, our algorithm contains a timeout mechanism to manage the computational resources , in addition to control the time spent on these operations, adhering to the time turn timeout of 2.0 seconds. This ensures the algorithm remains practical, particularly in real-time gameplay.

The function *timeout* is a boolean function that evaluates whether the time elapsed since the start of the program execution has surpassed a predefined threshold of 1.9 seconds. We set it to 1.9 instead of 2.0, providing some breathing room for steps in the algorithm that don't rely on the timeout function. This function is used in the more time consuming steps of the algorithm, such as simulation and expansion. In the case where the threshold is passed, the program terminates, returning the best move generated up to that point in the process.

2.3 Exploration Strategy Section

2.3.1 Overview

In our strategy, A* search and BFS complement each other to address distinct aspects of the exploration process. A* search takes the lead in providing optimal pathfinding, evaluating nodes based on cost and heuristic estimates. This ensures that the agent can strategically navigate the board, considering both offensive and defensive aspects in its decision-making.

While A* search excels in optimizing paths, BFS steps in to enhance adaptability within certain constraints. BFS systematically explores the state space without the stringent cost considerations of A* search, enabling the agent to identify potential moves within a specified maximum step limit. This collaborative approach allows the agent to balance between optimal long-term strategies and immediate short-term tactics, ensuring responsiveness to dynamic game conditions.

2.3.2 A* Search

In our exploration strategy, A* search evaluates potential moves based on cost and heuristic estimates, contributing to the strategic navigation of the game board. A* search allows our agent to explore nodes in the right direction towards the adversary, ignoring nodes placed too far away. This makes for an excellent offensive strategy that is efficient in time since it does not waste time searching through far away nodes.

Consider a scenario where the adversary is positioned at a considerable distance. A* search, by evaluating nodes based on cost and heuristic estimates, guides our agent to prioritize moves that lead closer to the opponent. This demonstrates how A* search plays a pivotal role in offensive decision-making when the opponent is strategically positioned on the game board.

2.3.3 Breadth-First Search

Breadth-First Search (BFS) serves a crucial role in our exploration strategy when the agent needs to address specific challenges related to adaptability and exploration breadth. While A* search focuses on optimal pathfinding using cost and heuristic estimates, BFS operates in scenarios where the exploration breadth is prioritized over the cost-effectiveness of individual moves, ensuring that the agent can respond effectively to dynamic game conditions. For example, in cases where A* search does not find a path or a defensive strategy should be prioritized, BFS comes into play to present a greater range of possible moves.

The algorithm starts from the agent's current position and explores neighboring states level by level all while incorporating heuristic estimates, ensuring that potential moves are identified within

the specified maximum step limit (*max_step*). This breadth-first approach enables the agent to identify potential moves efficiently.

2.3.4 Combination of A* Search and BFS

The heart of our exploration strategy lies in the *all_moves* function, which seamlessly integrates the capabilities of both A* search and BFS. This function serves as a comprehensive move generator, strategically selecting between A* search and BFS based on the specific requirements of the exploration scenario.

Within *all_moves*, A* search is employed to focus on optimal pathfinding and strategic navigation. It is used to identify the shortest distance to the adversary, and employed as our search strategy of choice if the distance is over *max_step*.

Conversely, when adaptability and exploration breadth are crucial, the *all_moves* function seamlessly incorporates BFS. This includes scenarios in which the distance between the two players is within *max_step*.

The dynamic selection between A* search and BFS within the *all_moves* function allows the agent to adapt its exploration strategy based on the immediate demands of the game. This collaborative approach allows our student agent to efficiently explore a diverse range of options, strategically navigating the board.

2.3.5 A* Search and BFS Heuristic Functions

Our algorithm heavily relies on heuristic functions to determine which moves to prioritize and simulate on. This is achieved within our A* Search and BFS by heuristic functions who dynamically adjusts the priority values, influencing the offensive and defensive nature of the agent's decisions. An offensive and defensive heuristic function is incorporated using two crucial heuristic functions, *calculate_distance* and *calculate_direction*, significantly contributing to the A* search process:

- *calculate_distance*: This heuristic calculates the Manhattan distance between the agent's current position and the adversary's position. It provides a measure of how far apart the two entities are on the game board, allowing the agent to prioritize from legal moves.

- *calculate_direction*: The offensive heuristic calculates the direction of the adversary relative to the agent's current position. It assigns the priority values 1, 0.5, or 0 based on whether the direction is towards the adversary, perpendicular, or away respectively. This heuristic affects the priority of the move, guiding the agent to make offensive moves that strategically position it in relation to the adversary.

The *calculate_distance* function who assigns a higher priority to moves that bring the agent closer to the opponent. This prioritization aligns with our offensive strategy. By favoring moves that advance towards the opponent, our agent applies pressure and creates strategic positioning advantages. The *calculate_direction* function indicates to us if a barrier is oriented in the direction of the adversary, the heuristic will give priority to positions where the barriers do. This is in the effort to impede the opponent's movements and limit their strategic options. Additionally, the algorithm favors moves that involve moving towards the direction of the barrier, enhancing the defensive posture and reinforcing strategic positioning.

Furthermore, the heuristic considers the number of surrounding walls, favoring moves with fewer adjacent walls, providing freedom of movement and flexibility to our agent. The heuristic calculation is comprehensive, combining offensive distance, barrier direction, and the defensive aspect of wall count. It is expressed as: $(1 - distance\_between\_players/20) * in\_direction\_of\_opponent * number\_of\_walls$.

Another priority function is implemented through a dedicated function called *adjust*. This function dynamically adjusts the priority values, influencing the offensive and defensive nature of the agent's decisions. The adjustments are tailored to enhance the agent's mobility and strategic positioning on the game board, they are determined by the number of moves left before and after placing a wall.

- If the number of adversary moves (*adv_moves_aft*) after placing a wall is less than the number of adversary moves before placing a wall (*adv_moves_bef*), it increases the priority value (p) by 0.5.

- If the number of agent's moves (*my_moves_aft*) after placing a wall is greater than the number of agent's moves before placing a wall (*my_moves_bef*), it increases the priority value (p) by 0.5.
- If *adv_moves_aft* after placing a wall is greater than *adv_moves_bef*, it decreases the priority value (p) by 0.5.
- If *my_moves_aft* after placing a wall is less than *my_moves_bef*, it decreases the priority value (p) by 0.5.

These adjustments dynamically reflect the agent's tactical considerations, where higher priority values are associated with moves that limit the adversary's mobility and increase our agent's flexibility, providing the agent with strategic advantages on the game board. All of these heuristic functions coupled together form the core of our agent's decision-making process during the exploration phase.

2.4 Simulation and Heuristic Refinement

In addition to the core exploration strategy, our agent employs simulation and heuristic adjustment to enhance decision-making. These components play a crucial role in evaluating potential moves and fine-tuning the agent's strategy. The simulation step is integral to our agent's decision-making process, allowing it to assess potential moves beyond a single step.

The *simulation* function is a key component of our agent's decision-making process.This function is responsible for simulating potential future game states by recursively exploring the consequences of different moves. It takes a set of potential moves generated by the *all_moves* function, representing the agent's possible actions in the current game state.

For each potential move, the function projects the game forward by temporarily applying the move, updating the game state, and recursively calling itself to explore subsequent moves. This recursive simulation continues until a predefined depth is reached or a terminal game state (win, lose, or tie) is encountered. A win has a score of 1, a loss has a score of -1 and a tie has a score of 0. The function keeps track of the cumulative scores achieved during these simulations. The scores represent the desirability of each move based on the simulated outcomes. The final output of the simulation function is the average score across all simulations, providing a measure of the move's overall strategic value under various potential scenarios. This value is stored at index 1 in its move's sublist. Thus, this Monte Carlo style simulation approach allows our agent to make informed decisions by considering a broad range of possible future game states.

2.5 Step Function

The *step* function orchestrates the entire decision-making process and is responsible for selecting the best move to execute in the current game state. It encapsulates the core logic of evaluating potential moves, incorporating heuristics, and simulations. The step function is invoked at each turn to determine the optimal move within the given time constraints.

1. Initialize Data Structures: The function starts by initializing the data structures necessary for decision-making. It creates a list of the top 10 potential moves by calling the *all_moves* function, which integrates A* search and BFS strategies based on the exploration scenario's demands.
2. Heuristic Adjustment: Before simulations, the function incorporates heuristic adjustments. Within the *all_moves* functions, the heuristic function: $(1 - distance\_between\_players/20) * in\_direction\_of\_opponent * number\_of\_walls$ is utilized to prioritize the list of potential moves. The step function then calls the *adjust* heuristic to fine-tune the prioritization of moves. These heuristics provide insights into the spatial relationship between the agent and the adversary, guiding the agent towards strategic positioning.
3. Simulation: With the set of potential moves and adjusted heuristics, the *step* function initiates the simulation process. It calls the *simulation* function, projecting the game forward by exploring potential future states. The simulation assesses each move's desirability by considering outcomes beyond a single step, providing a comprehensive evaluation of strategic value.

4. Select Optimal Move: After simulations, the *step* function selects the optimal move based on the cumulative scores obtained, and if there is an equivalence, it compares priority. Thus, the move with the highest strategic value is chosen for execution.

5. Time Management: The *step* function calculates the time elapsed since the program's start. By considering this elapsed time, along with the timeout function embedded in exploration and simulation, the agent ensures it adheres to the game's turn timeout of 2.0 seconds.

In summary, the step function represents the culmination of our agent's decision-making journey. The comprehensive evaluation process ensures that the agent navigates the game board with a balance of offensive and defensive considerations, responding effectively to both immediate challenges and long-term objectives.

3 Quantitative Analysis

3.1 Depth level on a size 12 board

The way we designed our algorithm, our depth is only 1 level, no matter the size of the board or the branch chosen. We only look at the possible children of the current position and we use the simulation scores in order to look-ahead further into the game. However, our simulations are able to look up to 4 levels of depth, prioritizing the branches with the highest heuristic score. So, the actual depth achieved should be 5.

3.2 Breadth achieved on a size 12 board

The maximum breadth we allowed for is 10 moves. However, at the beginning of a game for a size 12 board, since there are many moves available and it takes a longer time for our simulations to reach an endgame, the actual breadth reached is often 1 or 2. Furthermore, in the case where the adversary is very far from our agent, the breadth is also 1 since we decide to simply take the furthest point possible. Within the simulations, the breadth for both the max and min player is at most 3 at each level of play for 4 levels.

3.3 Depth & Breadth scaled with board size

As explain above, the depth explored is 5 no matter the board size. As for breadth, the allowed maximum is 10 no matter the board size, but the actual breadth reached has an inverse relationship with the board size, where it can reach the 10 allowed with a size 6 board and the 1-2 for a size 12 (when there are a lot of moves available).

A rough estimate of the big-Oh of simulation algorithm is around: $O(n * (b^d))$ Where n is the number of simulations, b is the branching factor and d is the depth of the A* search and BFS. We came to this conclusion since for each step, we first run our *all_moves* function which should be the complexity of an A* search algorithm: the branching factor to the depth. We then multiplied it by the number of simulations we run. These are the parts of our algorithm that take the most time and thus, we consider their time complexities as a rough estimate of the big-Oh of our entire algorithm.

3.4 Impacts of heuristics

The heuristics we implemented help immensely in terms of run time since we used it to determine which moves we want to consider and simulate on. Our heuristic evaluation function gives a priority for each possible next move, and we pick based on the top 10 moves, greatly reducing the number of children to consider. Comparing the depth, there is no difference, but the breadth does allow for many more simulations. Without the heuristic functions, our simulations would be the same as doing a random walk at each step, and even though the breadth allowed would be greater, it would be a lot less effective. We would have to let it look at all the children if we want the same results, but there is not enough time for that.

3.5 Predictions

3.5.1 Against a random agent

Our agent demonstrates a consistent 100% win-rate when evaluated against a random agent. We have extensively tested our agent against the provided *random_agent* and it has consistently resulted in victories in our agent's favour, establishing a high level of confidence in our agent's ability to dominate in this scenario.

3.5.2 Against an average human agent

In evaluations against an average human agent, we anticipate an 80% win-rate, an approximate number we concluded based on the games we have played against our agent, and the games we asked some friends to play. We asked people who were unrelated to computer science in general to simulate how an average player who did not know our agent's strategies and weaknesses would play.

3.5.3 Against our classmates' agents

Anticipating an 80% win-rate against our classmates' agents, this prediction is based on a careful consideration of our agent's strengths and weaknesses. Since we don't know how well our classmates' agents can play, we can only base ourselves off our agent's weaknesses and some testing we did while playing against our own agent.

4 Advantages and Disadvantages

One of the main advantages of our algorithm lies in the effective heuristic that's used in the decision-making process. The heuristic evaluation function considers offensive distance, barrier direction, wall count, and number of moves left, allowing the agent to make strategic decisions aligned with an offensive leaning stance. This proves to be a valuable asset, particularly when facing less-experienced players, like *random_agent*, as it provides a strong counter-strategy. Additionally, our algorithm uses the efficiency of A* search and breadth limited version of Breadth-First Search (BFS) in the *all_moves* function, producing an ordered list of potential moves based on the current game state. The effective exploration of potential moves allows the algorithm to allocate more time to other critical components, such as the simulation step. This enhances the algorithm's ability to consider a diverse range of possibilities, contributing to its adaptability.

On the flip side, there are some visible drawbacks to our algorithm. Notably, the randomness inherent from the simulation step, since our score depends on an unpredictable opponent player. This could lead to suboptimal decisions based on chance outcomes. Furthermore, the limit on exploration in the A* search and Breadth-First Search (BFS) process, where we only take the top 10 legal moves from the ordered list, constrains the algorithm's ability to thoroughly explore more moves, potentially missing out on better options that could lead to more ideal scenarios. Some expected failures include premature termination caused by the *timeout* function, leading to less simulations done, and inadequate heuristic tuning. We also lack the implementation of wall pattern matching, leading to a failure to observe patterns and when the agent is trapped within one. Moreover, we only explore one layer of depth in our tree, which can greatly reduce the ability to explore potential moves. These expected failures can cause our agent to make suboptimal decisions, resulting in a compromised overall performance.

5 Other Approaches

Our general idea to approach the project was always a Monte-Carlo Tree Search inspired algorithm, however, we had many earlier versions that were weaker than our final agent. For example, our first version contained no heuristic functions at all and would simply take a random step, and run random simulations on that step, taking random steps at each turn until the game is over. This version heavily relied on probabilities and pure luck, and it was quickly obvious that it was not the strongest of agents, only slightly better than the random agent. For the next few versions, we added a nested class as a way to organize our data structures easily, added a function that obtains all the possible next moves and

implemented a few basic heuristics such as, checking for an instant end game move and not putting ourselves into a space with 3 walls. Realising that these heuristics could be generalized, we designed a heuristic function and implemented it into the function that gets all the possible next moves as a way to prioritize certain moves over others. Testing these versions were much more successful against the random agent, but we noticed it was slow on the run time. We made the decision to get rid of the nested class and simply use a list data structure to contain our nodes as well as adding a few more offense heuristics such as the direction of the wall towards the opponent. Somewhere in the last few versions, we also tested out having our simulations be not so random, in other words, an approach inspired by the Minimax algorithm, where two players play optimally. It takes longer to run, but it gives better results than the completely random simulations.

Compared to our final approach, it was clear that earlier versions behaved a lot more randomly, and that with less heuristics, the agents were playing a lot less on the offensive. Our final heuristic functions greatly improved the way our agent is able to trap the opponent a lot quicker and also escape being trapped by itself or the adversary.

6 Improvements

There are many ways we can further improve our agent. The most obvious one would be to have an algorithm more similar to the actual Monte-Carlo Tree Search one, where many layers in the tree are explored and selection of nodes are based on a formula like UCB. We currently only have one layer of children nodes explored, which limits the depth of our predictions greatly.

Another main issue is the efficiency of our algorithm. The more simulations it can run, the more accurate our results would be. However, at the current state we do not have the time to run as many simulations as we would like to. Our results are therefore not as precise as they could be. We would need to change some things such as the number of times we run A* and BFS to make our algorithm more efficient.

Furthermore, we also lack the ability to recognize patterns in the walls of the game. This could be improved by introducing a data structure that will keep track of which walls are connected in the game.

Additionally, we also thought of ways to optimize our A* search and BFS algorithms in order to have a more efficient algorithm, which could save us a lot more time, enabling us to run more simulations and gain more accurate results.

Moreover, in general, our heuristic values need a lot of more fine tuning and testing. The weights of heuristic values inside our algorithms should be adjusted based on testing results in order to achieve a heuristic function as close to perfect as possible.

7 Conclusion

In summary, our journey in developing an intelligent agent for *Colosseum Survival* has led up to an algorithm that excels in strategic decision-making. Integrating A* search, Breadth-First Search (BFS), simulations, and intricate heuristics, our agent demonstrates an adaptive approach that balances offensive and defensive considerations.

Our algorithm's evolution, from early versions that are reliant on randomness to the heuristic-driven final agent, reflects our refinement process. The agent has reached a competitive level, consistently outperforming the random agent and providing a considerable challenge to average human players.

Despite its strengths, our agent does have more room for improvement. Enhancing algorithm efficiency, pattern recognition capabilities, and ongoing fine-tuning of heuristic weights are focal points for future iterations. As we aspire to polish our algorithm, the overarching goal remains the same: creating an agent that conquers challenges posed by adversaries in the dynamic realm of the *Colosseum Survival* game.