COMP 424 Final Project Game: *Colosseum Survival!*

Maxens Destiné maxens.destine@mail.mcgill.ca
Software Engineering Undergraduate
McGill University
Ryan Sowa ryan.sowa@mail.mcgill.ca
Software Engineering Undergraduate
McGill University

1. The 2 in 1 Agent

Our agent is a combination of the agents created by each team member. The agent will choose the Monte-Carlo tree search algorithm (*student_agent.py*) if the number of possible moves is below a certain threshold. If the number of possible moves is too large, the *Min-Max* agent (*helper_student.py*) will be chosen instead.

1.1 Reasoning Behind the Combination

The first team member who implemented his solution started by finding a way to beat the random agent at least 90% of the time. His reasoning was that once the agent could always beat the random agent, it could then be modified to achieve success in a *PvP* (player vs. player or rather student agent vs. student agent) scenario. The problem was that the agent searched over all moves which meant that while the search breath was large, the depth could not be increased easily. The agent was still successful against the random agent, but the possible modifications that could be added seemed to be capped by the time limit of 2 seconds. Following, the second team member implemented his solution and he used the first team member's agent as practice for his own. He implemented a Monte-Carlo tree search scheme which allowed his agent to beat the min-max agent as long as the search space was small enough. After evaluating the strengths and weaknesses of both agents (breath but smaller depth vs depth but smaller breath), it was decided that the best way to make use of both was to combine them. If the search space was too large for the Monte-Carlo to make a reasonable move, the min-max algorithm would be used. Otherwise, the Monte-Carlo would be used.

2. Min-Max Agent

The *Min-Max* agent can be found in the *helper_student.py* file. The logic of the agent is shown in Figure 1. This agent works by going over all possible steps and finding the best one given the current board state. The depth of search of this agent is very small. It will at best search 3 moves in advance, which happens when it is looking for a safe non-losing move:

- If I take this move...
- And the enemy takes this one...

- Can I still take a safe move?

Otherwise, it only checks if it can directly win. Another way to see this is that this agent

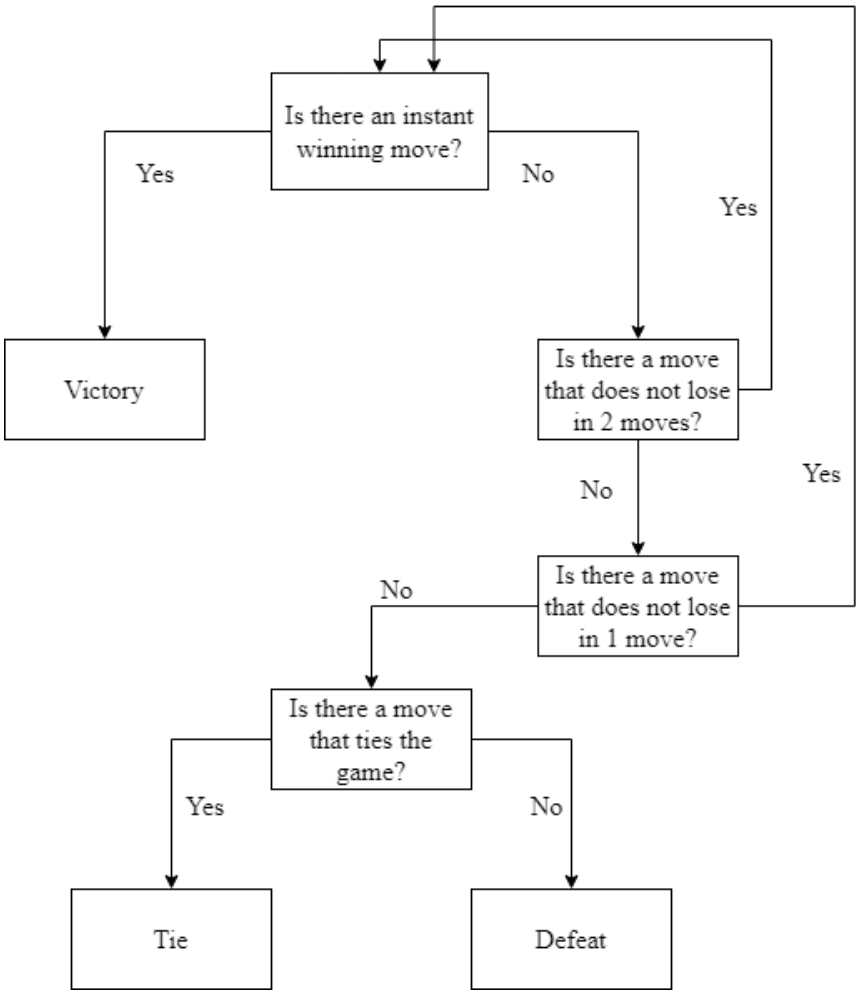

Figure 1: Logic flow of the min-max algorithm. A "yes" edge indicates that a move is taken by the allied agent. The arrows going back to the top refer to the state after the enemy played his move.

sacrifices depth in favor of breath. It guarantees that if there's a way to win instantly it will do so. It also guarantees that if there's no way to win but that there is a non-losing move, it will take it. The agent categorizes non-losing moves in 2 groups (winning moves are not included). The first group is of lesser priority than the second one and refers to the moves which, when taken, will not allow the enemy to beat the agent right after. In other words, those moves guarantee that there will be no instant defeat. The second group is like the first one, but it rejects moves where the opponent can win 1 turn later. This refers to moves where the opponent's body forces the agent to stay in a vulnerable position or to kill himself. Figure 2 gives an example that illustrates how the first group of moves may not lose instantly but will still most likely lose soon after because the agent is limited by

the enemy's body. While computing moves and other data, this algorithm regularly checks for the time limit. If the time limit is close, it will take a random move. It also catches exceptions and takes a random move when one occurs. This was done as an attempt to obtain the 5 bonus points.

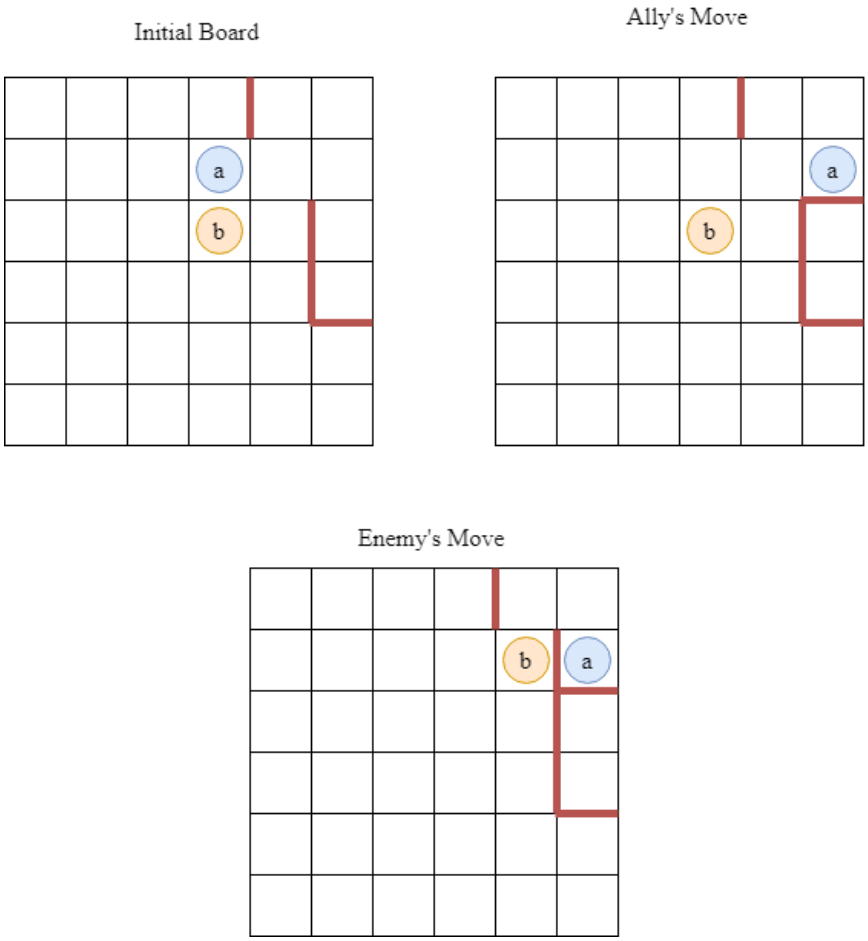

Figure 2: Example of a game scenario where a move results in a delayed defeat or a vulnerable position

2.1 Technicalities of the Agent's Name and Theoretical Basis

Although we call it the "min-max agent", we know that it is not really "min-max" as the entire search space is not explored (it would take too much time). The association with the min-max algorithm was made because when it is its turn, the agent will look over all possibilities (of a certain depth) and take the one which maximizes its chances of victory. In other words, if we imagine the min-max algorithm, the depth of nodes would vary between 1 and 3 edges. The value of each move would be the following: a victory would bring a 3 to the max root (our agent), a safe non-losing move a 2, an unsafe non-losing move a 1, a tie a 0 and a losing move a -1. Thus, at every turn, our agent is trying to maximize the

value of the move it takes. The actual implementation uses a list to express the priority of moves (best moves first). The first move in the list is the one that is returned. This priority may be hard to notice by only skimming the code, since in some cases, to reduce useless transfer of data, a list with only 1 move is returned. For example, if a winning move is found, the algorithm returns immediately as computing other moves is useless. Also, if a safe non-losing move is found after failing to find a winning move, the algorithm will again return immediately as a safe non-losing move is the best type of move that is possible in such a situation. A more appropriate name for this algorithm would be *Depth Limited Min-Max*.

2.2 Motivation for this Approach

The initial goal of this algorithm was to beat the random agent reliably (90+% win rate). Once beating the random agent became trivial, the plan was to modify the algorithm to be better suited for *PvP* situations. After playing around 10 manual games against the random agent, it was observed that the random agent often killed itself. It was thus decided that surviving would be prioritized as it would give enough time to the enemy agent to kill itself. Afterwards, it was observed that the random agent often put itself in vulnerable positions (1 move away from defeat). It was thus decided that the agent should take winning moves when possible, as this feature would also be useful when playing against other student agents. In summary, by observation, it was decided moves would be considered one of the following: winning, safe non-losing, unsafe losing, tie, or losing, based on the state of the board after taking them and that they would be prioritized in that order. *A case could be made for putting the ties before the unsafe non-losing moves.*

2.3 Comparison With Abandoned Approaches

Another approach was developed where in addition to the current feature of the algorithm, the agent would also look for moves that would allow him to beat the enemy in 2 moves. Similar to the defense feature which prevents body blocking, this one would help the agent find body blocking opportunities against the opponent. It was successfully implemented but had to be removed to ensure that time breaches would not happen (it is better to find a safe move than to make a random one because of a time breach caused by looking too much for a body blocking move). In brief, that approach performed better than the current min-max if not for the time constraint which forced us to remove it.

3. Monte-Carlo Agent

The *Monte-Carlo* agent operates using the following strategy:

While time has not exceeded two seconds:

1. Compute a list of all possible moves.
2. For each move, run a random simulation
3. If the simulation ended in a win, add 100 to the score of this move.

4. If the simulation ended in a loss, subtract 10000 from the score of this move.

5. If the simulation ended in a tie, subtract 25 from the score of this move.

This strategy ensures that simulations which result in losses contribute a very low score to a move(so this move will rarely be chosen), whereas simulations which result in wins will generally contribute a positive score to a move. After the time has been exceeded, the move with the maximum score wins.

3.1 Technicalities of the Agent's Name and Theoretical Basis

The *Monte-Carlo* agent is based on Monte-Carlo Tree Search. However, the *Monte-Carlo* agent only explores a depth of 1 - in other words, it does not run simulations on children with a depth greater than 1. In addition, the original Monte-Carlo Tree Search uses the UCB score of nodes to decide which node to run a simulation on next. The *Monte-Carlo* agent simply runs simulation on all possible moves from the current state in order.

3.2 Motivation for this Approach

The greatest motivation for this approach was its simplicity. The idea behind the *Monte-Carlo* agent was very straightforward: run simulations on all possible moves and return the move which wins the most and throw away the moves which lose. After writing a method to run simulations, all that was left was to run simulations on all possible moves for a set period of time and return the "best" move. In addition, because each simulation was essentially simulating two random agents playing each other many times and picking the best of these simulations, it almost guaranteed a win against the random agent.

3.3 Comparison With Abandoned Approaches

The minimax algorithm was considered, however, there were two factors which made this algorithm non-desirable:

1. The branching factor could be extremely large, especially when the board size is 12.

2. Because it would be impossible to simulate all possible games, an evaluation function would be needed. However, such an evaluation function is not intuitive to come up with.

The original Monte-Carlo Tree Search was tried, however, it was soon made clear that if the algorithm considered nodes at a depth of 2 and larger, an insufficient amount of simulations would be executed in a time period of two seconds to come up with a realistic answer.

4. Pros and Cons of the Combined Approach

4.1 Pros

- Never goes over the time limit

- Guarantees taking a legal move

- Guarantees taking a non losing step if one exists
- Guarantees winning if such a step exists
- Catches exceptions (including time breaches) and does a random move if they happen
- By choosing the *Min-Max* Agent or the *Monte-Carlo* Agent based on the number of possible moves, we are able to make the best decision with respect to depth and breadth

4.2 Cons

- Min-max can only see 3 steps into the future
- Min-max rarely attacks the other agent, mostly only tries to survive
- Monte-Carlo is only effective when the number of possible moves is small
- Neither Min-max or Monte-Carlo make us of the 30 seconds to make the first step of each rounds

5. Possible Improvements

A good improvement related to the implementation rather than the algorithms themselves would be to change the function which checks for the end of the game to only consider the zone in which the players are situated. If the game board's size was reduced during the game, it is useless to consider the area which the players cannot access when trying to determine a game's end. Another improvement would be to make the agent intentionally reduce the board size. The reasoning behind this is that the "killer" algorithm, the Monte-Carlo tree search, needs a smaller board to be successful. By reducing the board size intentionally, it would be possible to use the Monte-Carlo scheme sooner, which is better than simply hoping that the min-max scheme survives long enough. Finally, we could change the "weights"(the score that is returned) of losses and wins for the *Monte-Carlo* agent to achieve a better win rate.