COMP 424 Final Project Game: *Colosseum Survival!*

Student names & IDs: Eliška Suchardová (261118612) & Freja Petersen (261122938)
Due Date: Dec 5th, 2022, 8:59PM EST

1. Introduction

In this report, a game playing agent for the *Colosseum Survival* game will be introduced and discussed. The developed agent uses a Monte Carlo Tree Search (MCTS) approach to determine its moves. During the development of the code, it was quickly decided not to explore minimax or alpha-beta-pruning approaches since the branching factor of the game is simply too large. Therefore, this approach, where the explored branches of the game tree are hidden and random, was chosen.

In our implementation, the MCTS plays out a number of scenarios for each possible and valid move of the player. In the scenarios, both players make random moves. Thus, the result of using this search is a heuristic that tells the player, that making a certain move would be better or worse than another move. Since, in reality, none of the players are random, the heuristic is not perfect.

2. Theoretical background

Monte Carlo tree search [1] is a search algorithm utilizing random sampling. It is very often used in AI for games, where no reliable state evaluation function exists like Go or chess. The algorithm itself is divided into 4 stages:

1. Selection - Start from root and find a leaf node to be expanded.

2. Expansion - Create child nodes for possible moves from current leaf node (unless it is the end state) and select one of them to be played out.

3. Simulation - Let the game play out until the end from the selected node.

4. Back-propagation - Take the result and update the nodes from selected node along the path to root based on how the play out ended.

With no surprise, the move represented by the node with the best store is chosen. Generally, proper balancing between exploration and exploitation of the selection policy is one of the hardest part of implementing Monte Carlo tree search. However, for our solution, we decided to opt for a simplified version, where each possible move is guaranteed to be played out a number of times.

1. Source: Monte Carlo Tree Search: A Review of Recent Modifications and Applications `https://arxiv.org/pdf/2103.04931.pdf`

3. Development of agent

3.1 First version

The initial implementation of the game agent worked in the following way: when it is the agent's turn, 10 different scenarios of the game using random moves would be played for each possible and valid move. Then the move with the most wins would be chosen. In the example of figure 1, an example can be seen where the agent (player A) has 17 possible positions to go to, and for each of these, there are up to 4 directions to put the barrier. For each of these ≈ 60 valid moves, 10 different games would be played out, thus 600 games would be played out to the end for this specific state.

The strategy was constructed such that there was an outer loop, which was iterating over the possible moves, and an inner loop, which iterated over 10 different simulations. Thus, this construct was very rigid and would always run 10 simulations of each move. This resulted in a lot of computation time for each move, which exceeded the limit, when the board size was above 7.

Even though this version was very slow, it showed to work very well against a random agent, where it would win every time. But it would not be suitable for submission to the class tournament, since it exceeds the time limit.

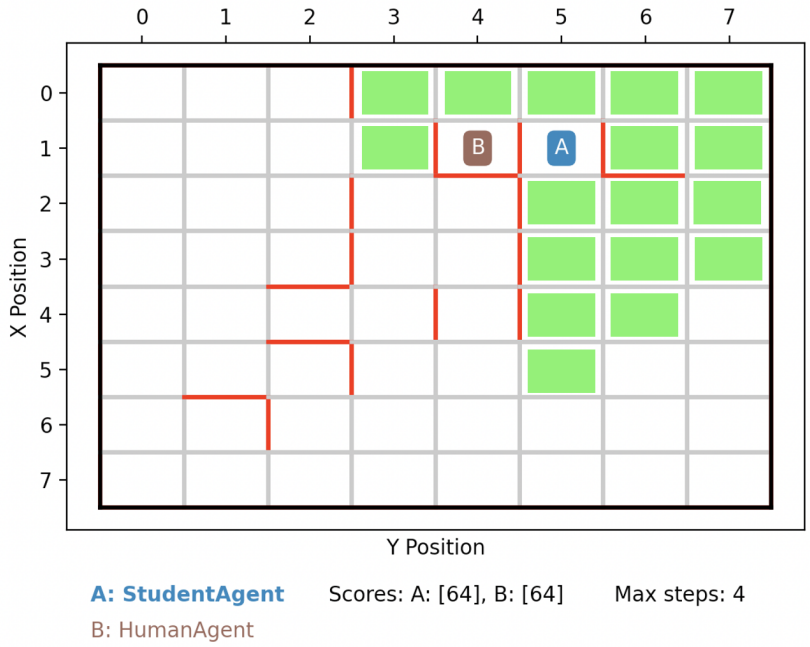

Figure 1: Example of a state of the game. The green boxes show possible positions of player A in the next move.

3.2 Second and final version

In order to improve time performance, the code was changed to allow for more selective simulation. This was done by inverting the nested loop in the initial version, such that the

outer loop iterated over the game simulations and the inner loop iterated over the possible moves. This structure is the final structure of the agent, but more improvements were introduced to improve performance, which will be explained in this section.

In this new structure, each possible move is tested once, then twice and then again. After the first two tests, only the moves with a score among the 25% best will be explored further. This selection of moves continues until there's only one move left or until the number of iterations exceed a threshold, which we will call `num_sim_MCTS`.

3.2.1 Tuning num_sim_MCTS

The parameter `num_sim_MCTS` is tuned depending on the board size. The tuning is done to take runtime into consideration. For a large board size it takes a very long time to run the simulations, since there is a large number of possible moves *and* the game can last many moves before it is over. Therefore, the agent should be tuned, such that it performs fewer simulations in the beginning of the game (low `num_sim_MCTS`) than later in the game. Fewer simulations means a less informed agent and thereby a more random move. This is okay in the beginning of the game, since it is less critical if the agent makes a bad move, because it will have time (moves) to save itself from its troubles. Later, though, it is wished for the agent to take more informed moves, and thus `num_sim_MCTS` should be higher.

This tuning is implemented in a way such that `num_sim_MCTS` is as large as possible without the agent exceeding the time limit of 2 seconds. This is done by measuring the time spent on the simulations (given some initial value of `num_sim_MCTS`), and if the agent spends more than 1.9 seconds, `num_sim_MCTS` is halved. If, on the other hand, the agent spends less than 0.3 seconds, `num_sim_MCTS` is doubled. The effect of this tuning is seen in figures 2 and 3, where the agent is playing against itself. Note that the y-axis is log transformed in figure 3. It can be seen that the agent spends less than 2 seconds on most moves (except the first one, which will be explained shortly). If it exceeds the limit, it will correct `num_sim_MCTS` and be under the limit in the next move. Figure 3 shows that the number of simulations performed per move is small in the beginning and increases as the game progresses.

3.2.2 Pre-calculations

In order to further improve time performance in the beginning of the game, some pre-calculations were introduced. These pre-calculations simply compute forward in time what the evaluations of future moves would be. Say, the result of doing the MCTS is that the player should make move m_1. Then, another instance of the MCTS is made from this position, yielding the move m_2. This is done a certain number of times, given by the parameter `num_precalculated_steps`. The code is tuned, such that the player never - on our machines - spends more than the 30 seconds doing these pre-calculations before the first move.

After the agent makes the first move, the opponent will make a move that is most likely not the same as the one expected from the pre-calculations. This means, that when the agent makes the second move, this might not be the best move in the real game. Though, in the case of a large board size, the agent will not have time to make an informed decision (`num_sim_MCTS` is probably 1 or 2), so in this case, the move from the pre-calculations might

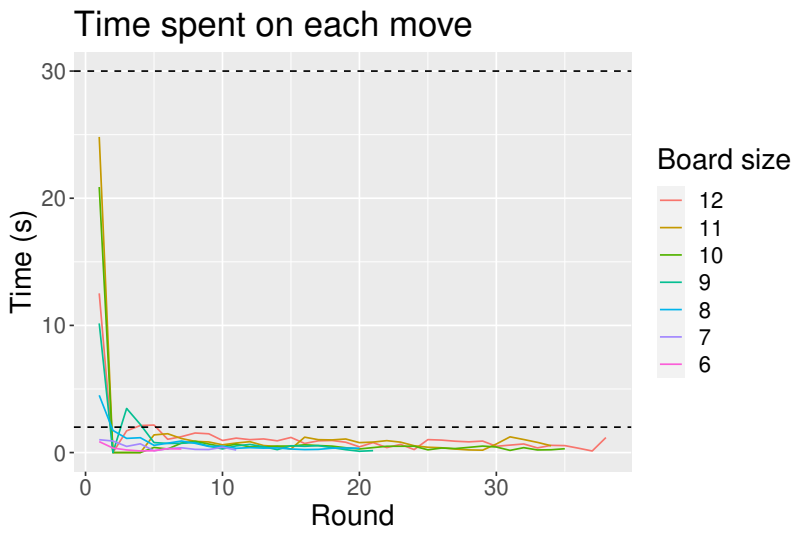

Figure 2: Time spent on each move

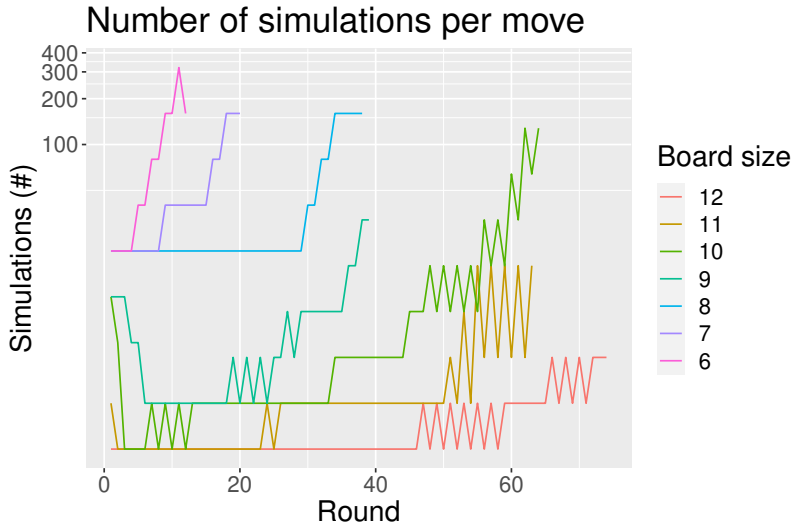

Figure 3: Number of simulation branches for each move (log scale)

be better. In any case, `num_precalculated_steps` never exceeds 5. If the pre-calculated move turns out to be invalid, it will be disregarded.

4. Discussion of agent

In terms of game performance, the agent beats a random player regardless of board sizes. When playing against a human player, it also performs very well on the smaller sized boards and beats the human player, but on the larger boards the agent does not always win. This is because of the main disadvantage of the player: the assumption in the heuristic about

the opponent being a random player is too weak. This results in the agent only being really good at planning a few steps ahead, whereas a human player is able to plan many steps ahead. This is not seen when the board is small, because in that case it is difficult for a human to plan ahead due to the limitations of the board. But on the larger board, the human can more easily plan to, e.g. make a wall down the middle of the board, which is almost impossible for the heuristic to predict.

Another disadvantage of the implementation is that the computation time is large. It is seen in figure 3 that for the largest board size, there are almost no simulations made, which means that the decisions are close to random. The available moves are still compared, but when it is only based on one simulation, the comparison is mostly based on luck (though, it is still better than a random agent). Thus, the method does not work well for large boards, when it has to comply with the time restrictions.

The main advantage of the agent is that it automatically updates the amount of simulations, such that it makes more informed decisions towards the end of the simulation. This is a big advantage in this game, since it is exactly in the end of the game, the most crucial moves are made. For example, in the case in figure 4, the agent (player A) should know that it is better to move to position (0,3) than for example (0,2) in order to avoid losing.

Another advantage is the simplicity of the heuristic, which makes the evaluation faster than a more complicated heuristic. There is a trade-off between the quality of the heuristic and the computation time. With the simple heuristic, we can perform more simulations (more exploration), whereas with a more advanced heuristic we would perform fewer but possibly better simulations (more exploitation).

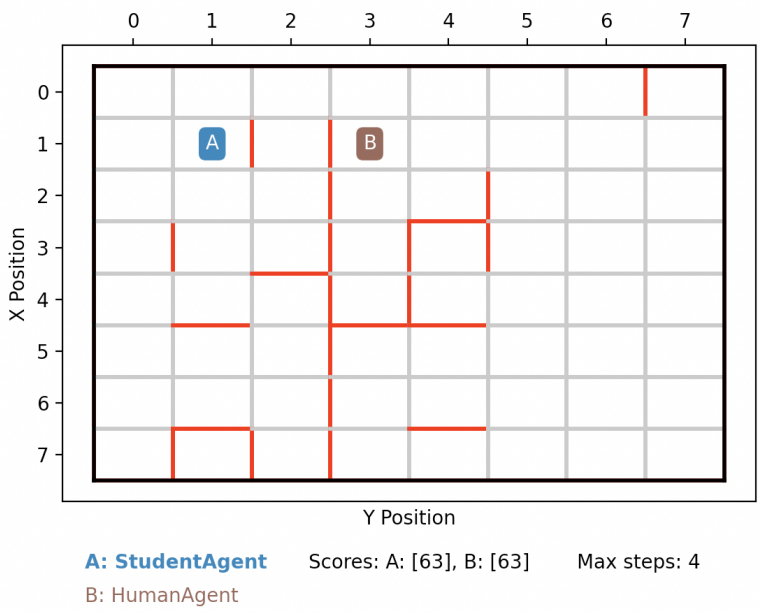

Figure 4: Example of a state in the game.

5. Future work

In the current version of the agent, it will not always choose to go to position (0,3) in figure 4, because it might simply not have enough time to compute the steps. Therefore, a way of improving the performance of the MCTS in terms of time could be to favour some steps higher than others when doing the simulations. This could be done by the following procedure: before doing the MCTS, check whether the game could be ended in the next move by the opponent, for example by performing the move (0,3,left) in figure 4. If so, only simulate games with the moves that places the player on the other side of this final move, in this case, (0,3,[l,r,u,d]) or (0,4,[l,r,u,d]). Preferably, the agent would take the (0,3,left) step itself to ensure that it is on the correct side of the barrier, and also to make sure that the game will be shorter by making the board smaller. This could be introduced in the agent as a strict policy or as a policy with a skewed sampling distribution. This update could improve performance in terms of both time and winnings.

Another improvement that could be introduced is to spend the first 30 seconds making a plan or policy based on the initial barriers. This could for example be by favoring the steps that connects the new barrier to one or more existing barriers in the MCTS and placing itself on the correct side of the barrier. This update could improve performance in terms of winnings, but not necessarily in terms of computation time, since it will not exclude any steps.

It would also be interesting to investigate the relationship between exploration and exploitation further. This could be done by starting with a simple heuristic and introducing a more complicated one towards the end of the game.

6. Conclusion

In conclusion, the implemented agent utilizes a simple MCTS approach that ensures a win against a random player and does a good job in challenging a human player. The agent looks forward to finally competing against worthy opponents in the class tournament.