COMP 424 Final Project Game: *Colosseum Survival!*

Mohanna Shahrad, and Melissa Katz

1. Introduction

The field of computer games is one where the need for computer agents is obvious, since it eliminates the need for two real players in a two-player game, such as in the case of the Colosseum Survival game presented to us. However, it is not sufficient to simply implement an artificial agent that selects moves randomly, since randomness does not accurately reflect human behaviour. In the case of our game, in particular, it often results in the agent just walling itself in, which prevents the human player from feeling as though their victory was earned. In this report, we'll investigate a way to improve the performance of the computer agent with the help of Monte-Carlo Tree Search, in order for the adversary to present an actual challenge for the human player.

2. Theoretical Basis

In class, most of our discussion on adversarial play focused on tree-based algorithms, like Minimax, Alpha-Beta pruning, and Monte-Carlo Tree Search (MCTS for short). Alpha-Beta pruning would not have been helpful in our situation, since we need to define a strong evaluation function for it to be effective. Since we had no prior knowledge of the game, it would have been next to impossible to find one that would be useful to us. MCTS is the exact opposite in this regard, since one can obtain excellent performance from it "as long as we know the rules of the game" [5]. Minimax is also an algorithm that would not need an evaluation function as long as we built the entire tree structure. However, that would not have been possible in our case, due to the branching factor of our game: On a 12x12 board, the branching factor is, on average, over 200, depending on the starting location of the agents and the placement of the initial walls (see Figure 1). For reference, Go has a branching factor of around 361 [5] and Chess has one of about 35 [1]. For that same board size, games can also go on for over 100 moves total (although the branching factor does decrease significantly over time), which means that there would be approximately 200^{100} nodes in the search tree, much more than the estimated 35^{100} nodes in the case of Chess. Considering our strict time limit, it would have been infeasible to build the entire tree and then run Minimax on it.

Taking all of the above into account, we decided to create an agent which runs Monte-Carlo Tree Search in order to decide its next move.

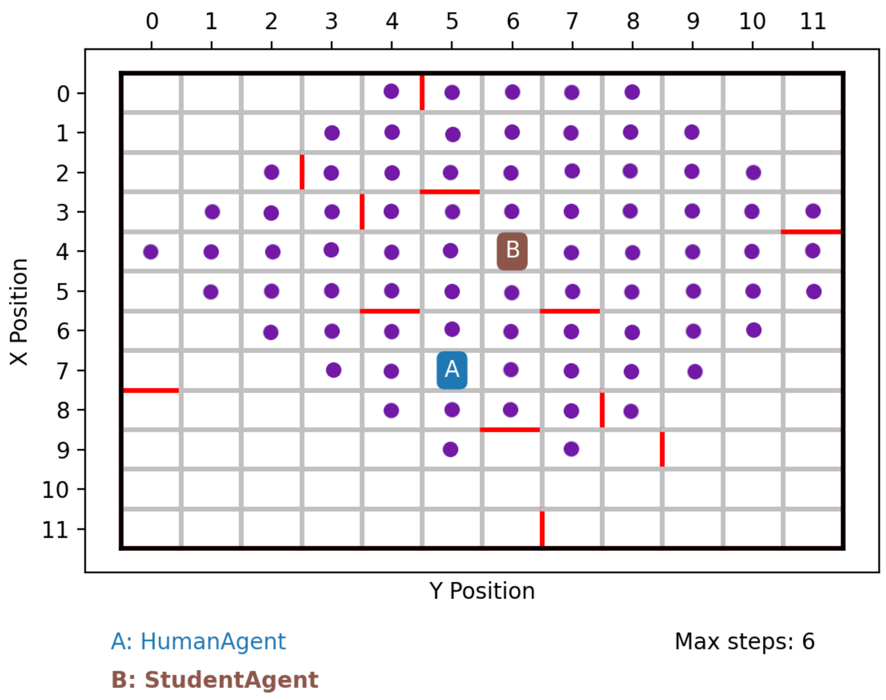

Figure 1: An extreme example of the high branching factor on a 12x12 board. The purple dots plus the initial state represent all the moves available to agent B. This totals to 291 different states.

2.1 Sources

We based our code partially on two GitHub repositories: *MonteCarloTreeSearch-TicTacToe* by user marshallgrimmett [4], for the general structure that our code needed to have, and *guided_monte_carlo_tree-search* by cwiz [2], for verifying correctness of the general flow of our algorithm (i.e Selection, Expansion, Rollout, and Backpropagation). To better understand how to implement progressive bias, we also consulted a paper by Marc Lanctot et al. [3]

3. Explanation of Approach

The bulk of our code is located inside the MCT_search.py file. In this file, we define two classes: Node and MCT. Node holds all the essential information about the state of the board (such as what move led us to this situation in self.move, and which agent made that move in self.turn). It also maintains relationships between nodes by keeping track of a node's parent and children in the search tree. Finally, it holds statistics that will be useful for selecting the best child, such as the number of wins and the number of visits, but also a value that defines a heuristic "worth" of the current state, depending on the number of walls that surround the agent (the more walls, the lower the worth). The motivation for this heuristic was to discourage moving to positions where it could be vulnerable (such as where it is surrounded by three walls). We also found that this technique helped the agent escape dead-end situations (see Figure 2)). On the other hand, MCT holds static information about the world in general, such as the size of the board, the maximal step size, or the time allotted for our search. It also holds the root node of the tree as well as a reference to the node we are currently examining when we go down the tree.

Whenever our agent must pick a move, it calls the run_tree function. In it, we first initialize the root of a new tree and calculate its worth. We then run a series of selections, expansions, rollouts, and backpropagations as long as we are below a certain time threshold (29.8 in the first step and 1.9 in all others).

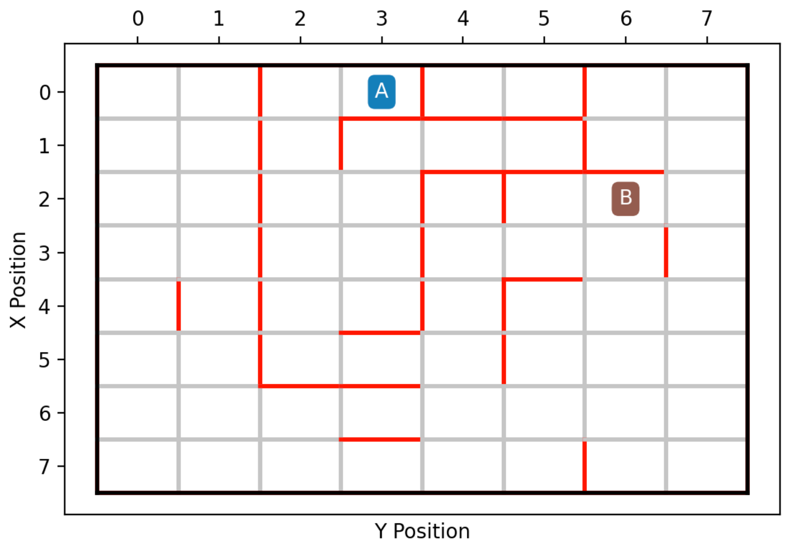

Figure 2: Agent A is stuck in a dead end. It must play optimally to escape.

3.1 Selection

Our agent starts out by picking the best leaf to explore using our tree policy. With the root as the starting node, we iterate through each of its children (until there aren't any) selecting the child with the best value according to the following formula:

$$\begin{aligned}
\text{Value}(n) &= UCT(n) + \text{bias}(n) \\
UCT(n) &= \frac{n.wins}{n.number\_of\_visits} + 2\sqrt{\frac{log_{10}(n.parent.number\_of\_visits)}{n.number\_of\_visits}} \\
\text{bias}(n) &= 0.05 \times \frac{-n.walls}{n.number\_of\_visits}
\end{aligned}$$

To counteract divisions by zero (when a certain node has never been visited yet), we give it a default UCT value of 1. Note that the above formula also takes into account the heuristic worth of the children, which makes positions with few walls around them more likely to have a better UCT. When we can no longer select another node, we start expanding it.

3.2 Expansion

Our best leaf now gets analyzed more thoroughly. If the leaf does not represent an endgame situation, we create a child chosen from every possible action it can take from the current position. While we determine the possible actions, we also work on putting in place a move heuristic, an assumption that moving towards the opponent will generally be favourable. This approach was decided through our analysis of the game's environment: indeed, if an agent retreats, it will be able to control only a small area in the next round, allowing the opponent more opportunities to restrict it more and more in that area, eventually leading to a win. If it is close to the opponent, on the other hand, it has more opportunities to put down walls that will help it win.

We thus document all the different positions our agent can reach. We also perform one static check: if the opponent happens to be encircled by three walls, we will check if the current position is the one needed to definitively close off the adversary. If yes, then we return that position only. If not, and the agent still has steps left, we will analyze each cardinal direction which is not obstructed by a wall. We also determine in what direction the move takes us, and we put the move in a different list depending on whether it takes us towards the adversary, in the diametrically opposite direction from the adversary, or in a "neutral" direction. Finally, when we have found all the positions and walls the agent can place, we do the following: if it's our agent's turn to put down a wall, it chooses a non-uniformly random move, meaning that it is more likely to pick a move that moves it towards the adversary than one that makes it retreat. If it's the adversary's turn, it picks a uniformly random move. We can now proceed to playing until we reach an end state.

3.3 Rollout

Rollout is quite similar to expansion. We keep following the policy described above: as we go down, we look at all the possible actions, and unless there is an obvious win situation, we pick a random choice. If our agent is choosing, the random move has a higher likelihood than normal of being one that takes us towards the adversary, while the other agent makes purely random decisions. Finally, when an end state is reached, we count up the number of tiles each agent controls and we set a score according to the results: 1 for our victory, -1 for our loss, and 0 for a tie.

3.4 Backpropagation

Now that we have the score of our decisions, we have to relay this information back up our tree. As we go up to the root node, we update the win information of the node as well as increment the number of visits. As we send our score further up our tree, we also flip the score value since a victory for a node representing the choice of our agent means that the adversary's choice in that node's parent resulted in a loss for it. Since we don't know whether the victory or loss was a result of our decision or of our opponent's, we have to flip the score.

3.5 Wrap-up

When the time allotted to MCTS runs out, we look through the children of the root and select the best one based on the number of visits. This is because the child with the best UCT value will be visited the most, and it in turn has the best UCT value because it has the most wins.

4. Advantages and Disadvantages of Approach

The main advantage of our approach is due to the nature of Monte Carlo Tree Search: we do not need expert knowledge of the game to create an effective AI agent. This also means that, if the rules were to change in the future (for example if we were to allow moves without forcing the placement of a wall) our algorithm would need minor adjustments instead of a major revamp. On top of this, thanks to the move and worth heuristics, our agent is encouraged to adapt more human behaviours (such as avoiding enclosed spaces or moving towards the opponent to cut off their access to tiles), which reinforces the illusion of intelligent behaviour.

However, the algorithm has flaws, the most evident being the same flaw as any approach using MCTS: it will often miss the optimal play because its random rollout strategy failed to choose it. As such, in situations where it has a single tile to place down to win the game, it might either win a few rounds later or it might actually end up losing. We also suspect that this randomness is what causes our agent so much trouble in near-enclosed spaces, despite the worth heuristic: it sometimes just fails to find the few moves that will lead to its escape.

5. Other Approaches

Although we never attempted to implement an algorithm that differs from our final one in a radical way, we nevertheless experimented with keeping the same tree throughout the game instead of restarting from scratch at every turn. We were motivated by the desire to reduce the number of repeated computations: for example, there could be situations where the agent starts at node $\boldsymbol{A}$, then goes to $\boldsymbol{B}$, which itself goes down to some end state $\boldsymbol{F}$. If the agent then decides that $\boldsymbol{B}$ is its best choice, then there could be a risk of it once again reaching $\boldsymbol{F}$ and redoing all the computations, despite there being no changes in the evaluation of this path. If we were to maintain our tree, this situation would be avoided.

To achieve this, we did the following: We knew that our tree was always guaranteed to create a node for each possible move from our current position. So, when we selected a particular move, we defined the root as whichever node $\boldsymbol{N}$ held that move. At our agent's next turn, we would then scan through the children of $\boldsymbol{N}$ and move the root once again if we managed to find the move that was selected by the opponent. If not, we would create a completely different root and run as normal. However, we eventually decided against this approach, since we realized that in the majority of cases, $\boldsymbol{N}$ would have very few children (when compared to the amount of possibilities), which meant that the move actually selected by the opponent was rarely among them. This meant that we were more often than not wasting computational power before starting a new tree anyway.

6. Improvements and Conclusion

In the end, we successfully implemented an AI agent that uses Monte-Carlo Tree Search to find a play that poses an actual threat to its opponent, even though it doesn't always find the optimal play. This was achieved with a combination of worth and movement heuristics as well as some static checking that allows it to win in very obvious situations. There is much room for improvement in order for our algorithm to act even more rationally: first, when we decide which move to pick with a non-uniform distribution, it would be interesting to test a wider combination of values in order to find the ones that make our agent behave the best. The above also applies to the constants that we use when calculating UCT values. Next, we could explore some algorithms that could augment the basic search: for example, it would be quite useful to implement a pathfinding algorithm in order to counter our agent's difficulty to deal with near-enclosed spaces. We could also expand its catalog of static checks: for example, we could make it check for the endgame as it's discovering its possible actions instead of for when it performs the random playouts. Of course, this could then cause a problem with the allotted time, reducing the number of simulations we can do within 2 seconds. Finally, if we were to expand this problem beyond the limitations imposed by the assignment, we could also explore the possibilities offered by the field of Machine Learning, such as neural networks. This would probably be the best version of our algorithm; after all, AlphaGo was able to beat professional Go players thanks to its Monte Carlo Tree Search algorithm augmented with a neural network.

References

[1] Jackie CK Cheug and Bogdan Mazoure. Comp424 - artificial intelligence. monte carlo tree search, 2022.

[2] User cwiz. guided monte carlo tree-search. *Github*. URL `https://github.com/cwiz/guided_monte_carlo_tree-search/blob/master/monte_carlo_tree.py`.

[3] Marc Lanctot, Mark H. M. Winands, Tom Pepels, and Nathan R. Sturtevant. Monte carlo tree search with heuristic evaluations using implicit minimax backups. URL `http://mlanctot.info/files/papers/cig14-immcts.pdf`.

[4] User marshallgrimmett. Monte carlo tree search - tictactoe. *Github*. URL `https://github.com/marshallgrimmett/MonteCarloTreeSearch-TicTacToe/blob/main/mcts.py`.

[5] Stuart Russel and Peter Norvig. Artificial intelligence: A modern approach (fourth edition).