COMP424 Final Report

Luc Baier-Reinio, Matthew MacDonald

November 2022

1 Agent Explanation

We implemented our agent using a Monte-Carlo Tree Search. Our agent maintains a Monte-Carlo Tree such that for every move we continue to expand its depth and accuracy by running more simulations. We use this tree for deciding on our moves, and build our tree using four main concepts (ordered as they proceed in our algorithm): Selection, Expansion, Simulation, and Back-Propagating. Note that in our code base, the majority of our logic is in the MCTSNode class.

1.1 Selection

In the selection phase, we start from the root node of the tree and explore the tree by choosing the most suitable children. Once we reach a leaf node, the selection phase ends. From a parent, we pick the child that has the maximal output of this function: f(child) = child.winning-percentage + $\mathbf{C} \cdot \sqrt{\frac{log(parent.times-visited)}{child.times-visited}}$. In the first term, we consider the winning percentage of the child (exploitation factor). Inside the square root in the second term, we consider how frequently we have visited this child, such that it more likely that children that have not been visited as much get chosen (exploration factor). We have an additional parameter **C** that scales how much we consider the exploration factor in this calculation.

The parameter C for a node is calculated with the expression $C = e^{\frac{1}{depth}} - 1$, where *depth* is the depth of the given node. The intention with this calculation of the C parameter was to incorporate the concept of simulated annealing into our MCTS [1]. We wanted to reduce the value given to exploration as more child nodes are expanded and simulations are run. This forces our algorithm to prioritize the exploitation factor as it descends deeper into the search tree.

1.2 Expansion

The expansion phase follows from the selection phase, where we expand the children of the node that was selected. In expansion, we add children nodes

representing all possible moves the player could make from this game state. We find all possible moves in a BFS depth limited fashion, expanding all moves n steps away from the starting position, before expanding all moves n + 1 steps away, and stop exploring the BFS algorithm further once n = max_steps. Note that in our code, there are additional helper functions that allow us to determine if we have a valid move from one cell to the next, if we can place a barrier in a given position, etc. We use these helper functions when expanding all children to ensure we are only considering valid moves when expanding.

1.3 Simulation

The simulation phase is performed on all nodes that were added in expansion. For each node, we simulate game(s), starting from the current game state that the node represents. Once the game is finished, (we check this using the check_endgame method from world.py) we return the result to the node (whether it was a win/loss/tie for the agent). Moves in the simulation are performed at random, and for choosing a random move we incorporated similar logic as the random move method in world.py.

1.4 Back-Propagation

Once we have the result from simulating a game on a node, we back propagate this result from the node up to the root of the tree. We keep track of a nodes parent, which allows us to propagate back up to the root node.

1.5 Timeout Logic

We were required to implement our agent with a 2 second time limit on its step function. This posed a challenge in both implementing the timeout logic accurately as well as in optimizing our agent enough to be able to determine an acceptable move in that short time. To begin, the root node of our search tree contains the time that is was initialized. This value is passed on to all child nodes so that we may continuously compare it with the current time to ensure that no more than 2 seconds have passed. The time is checked in two places, first in the simulation running method where a random move is chosen for the player, and second in the expansion phase of our search. If we are close to exceeding the time limit, we stop exploring and/or simulating, and return.

1.6 Memory Use

Another requirement for our agent was that it use no more than 500 MB of RAM. To ensure we did not exceed this amount, we used the *memory-profiler* package in Python to profile our step function. The memory profiler indicated that our step function call uses no more than around 90 MB of RAM. We anticipated a sharp increase in RAM used when we increased our number of simulations on each node, but our testing indicated that this was not the case. We also

expected significantly higher memory use on the first move, where the agent is allowed to compute for 30 seconds compared to the regular 2 seconds; however, it also did not effect the memory usage all that much. Further improvements in memory use could be achieved by using a language without automatic memory management.

1.7 Max - Min Nodes

In our MCTS tree, we have min and max nodes, representing whether it is the agents or opponents turn. The motivation for this is how nodes are selected in the selection process. When exploring the pre-existing tree and selecting nodes, we handle selection differently depending on who's turn this node represents. If it is the agent's turn, then for the exploitation factor we will consider how well the agent has done in simulations that were performed from this move. Conversely, if it is the opponents turn, its exploitation factor will be based on how poorly the agent has performed in simulations from this move. It is necessary for the exploitation factor to be computed this way. If the exploitation factor were always based on how well a move seems for the agent, then at min nodes, very bad moves for the opponent would most likely be selected. That would clearly not be ideal, because then to the agent a move may seem very promising, but only because it is considering very bad opponent moves in response.

1.8 Putting it all together

We have a build_tree method() that does most of the heavy lifting and puts all the pieces of the above logic together. In this method, from the root, we select a node, expand said node, perform simulations on its children, and back propagate the result back to the root node. We repeat this process until we are close to the 2 second mark, at which point we return. We call this method in the step() function on the root node, and once it returns we select the child of the root node with the highest winning percentage. However, we have some additional logic in the step() function that allows us to re-use the tree we have built up thus far. For our very first move, we create and call build_tree() on a new root node. Once we find the best child node for our first turn, before making our move, we set this to be the new root node. We do this because the node corresponding to the move we selected represents the game state after we made our move, and thus the children of this node represent the board state after all possible opponent moves. Then, on subsequent calls to build_tree, given that we have a node that represents the game state after the next opponent move, before our next move, we set the root node to the node that represents the game state after the opponent moved. This scenario is represented in the figure below:

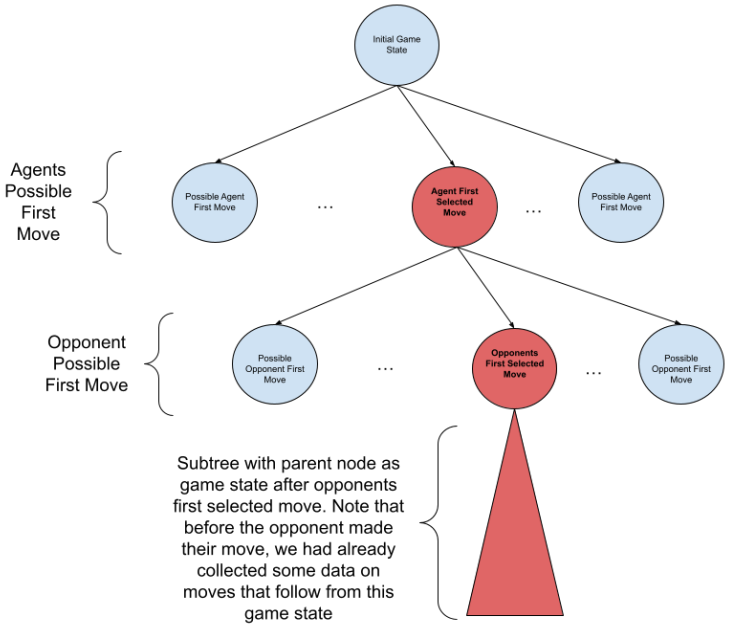

Figure 1: Tree visualization

The figure illustrates that after we make our first move, one of this node's children represents the game state after the opponents move. As illustrated in the triangle representing the subtree from the opponents move, we have also likely further expanded this node and already have some data that can help us choose a countermove. So if we have already built the tree at least once, on subsequent moves, we find the node that represents the current state of the board after the opponents move, set this node to be the new root, call build_tree on it, and return its best child. This is an optimization because as the game progresses, we have more data (more simulations ran) to help the agent make a more informed decision on better moves.

1.9 Agent Testing

While we are able to test our agent against ourselves and the random agent, after a certain point the information we gain starts to decrease. Once our agent reached a level of always beating the random agent, we ran into a roadblock of how to adjust our parameters to maximize the performance of our agent.

In particular, we were uncertain about how many game simulations should be

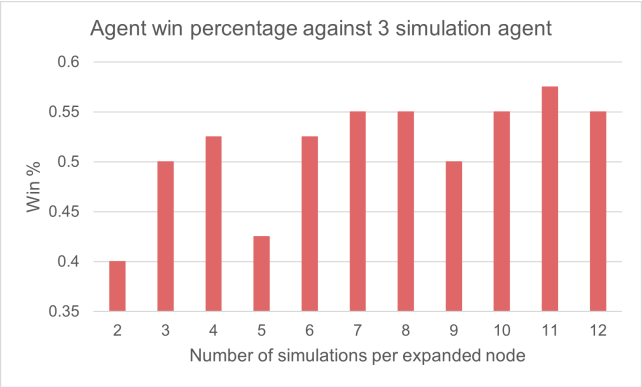

Figure 2: Bar graph displaying testing data

run on each node. We had arbitrarily set it to 3 during development, as it allowed us to run the agent without too much computational power. A concern we had about increasing the value was that too much time would be spent on computing the numerous simulations and not enough time would be spent on exploring a range of child nodes. To test this, we first modified the provided API. We added an additional parameter to the command line inputs that would assign a value to the number of simulations in our agent. We then decided a range of 2 to 12 simulations would be acceptable to test. It was also decided that we would run our agent with varying number of simulations against an identical agent running 3 simulations, unchanging. We wanted a significant enough sample size to try and reduce uncertainty as much as possible, but had to balance this with our local computational resources. We settled on running 50 games at each interval, for a total of 550 games. The results of these simulations are displayed in Figure 2.

Our testing showed that an agent running 11 simulations per node had the highest win percentage against an agent running 3 simulations per node, with a win percentage of 57.5%. Based on this data, we elected to run 11 simulations per node in our submitted agent. Future improvements in testing could be made by running more simulations, and simulation against agents employing different strategies (minimax with alpha beta pruning for example).

2 Theoretical Basis of MCTS & Motivation

At a very high level, our agent models playing the game as a search problem, where the states are board states, edges represent allowed moves, and the goal state is a board state such that our agent has won, or is in a winning position. We search and expand the tree in order to find winning positions, and choose a move that seems to maximize the chance of being in a winning position later in

the game. We have seen that search algorithms do in fact work to find a goal state, given that the agent understands what a goal is. Therefore, it was natural that we chose to model the gameplay in this fashion. However, the presence of an opponent makes the decision problem somewhat more complicated than the search problems [2]. We need to consider gameplay by the opponent, and therefore, we introduced the idea of min and max nodes to our MCTS algorithm. The motivation behind this is that we would like our agent to do well against competent players, and thus through min nodes, we try to consider good opponent moves to address this.

Moreover, in the selection phase, we balanced exploitation and exploration in our consideration for selecting a node. In class, we have seen the idea of exploration vs. exploitation in different applications, and have empirically seen how changing the emphasis on either of these two factors affects the optimality of Bandit algorithms. To incorporate the balancing of these two factors in our algorithm, we created a formula that includes a simulated annealing component. Given that the number of moves in the game is finite, we want to put less emphasis on exploration as the game progresses. We are able to have more confidence in our data as we get deeper into the game, as we have collected more information. Additionally, we believe that running simulations from a game state resulting from a potential move is a sensible decision. Simulations are not very computationally intensive due to their random nature, so it is easy to perform many of them on each turn. The large number of simulations mean that even though the moves in the simulation are entirely random, the data we get from these simulations will be at least somewhat accurate due to the high number of simulations run. The result of these many simulations will let us determine the 'best' move forward from the data we collect.

3 Advantages and Disadvantages

The power of MCTS in our implementation comes from the fact that random simulations are very fast to compute, so we can simulate a lot of random games in the 2 seconds we have to make a move. Given the sheer number of simulations we make, even though the moves are entirely random, the agent gets a good idea of what a good move to make is. It is also advantageous because it requires no use of heuristics while still being a very good agent. Algorithms like a-b pruning heavily rely on the use of heuristics to come up with a good ordering of moves to consider such that a lot of nodes can be pruned. These heuristics can be difficult to formulate, and require a good understanding of strategy in the game. Given that we do not have an 'expert' on this game to suggest effective heuristics, it would be hard to create a good a-b pruning algorithm since its success is dependent on and influenced by the quality of the heuristic. A third advantage of our algorithm is that we are able to reuse the tree structure from move to move, allowing the agent to become stronger and make better moves as the game progresses.

A shortcoming of our algorithm may be that when the board is very large, there are a lot of moves to consider at each game state, and given the two second constraint, we may spend too much time finding all the possible moves from a state of the board. This takes computation time away from running simulations, and our agent's success is heavily dependent on the number of simulations that we run. When the board is small, this seems to be less of an overhead, but as it grows, we get exponentially many more moves to consider at any given game state. An a-b pruning algorithm may be advantageous in this sense as if it has a very good ordering of moves, it may be able to prune a lot of sub-trees, and at each game state, potentially consider less moves. It seems natural that the fewer simulations we make, the less reliable our data is, and our agents success will suffer as a result. Additionally, as mentioned previously, our agent is weaker initially and gains strength due to it having more data as the game goes on, so this may be problematic for our agent as they navigate the opening of games.

4 Other Approaches

We initially considered a minimax or a-b pruning approach, however the computational overhead and reliance on heuristics and evaluation functions was worrying for us. If we didn't come up with an evaluation function, then we would have to exhaustively build the entire tree for every single move. Additionally, if we did not have a good heuristic, we may not be able to order nodes in such a way that we are able to prune a lot of subtrees. As we are not 'experts' at this game, this type of agent did not seem like it would be a good idea.

In an initial broken solution, we had no use of min and max nodes, and hence we ran into the situation where the agent would think a move is very good, however it only seemed good for the reason that it ran most of its simulations on a very bad countermove by the opponent, since it was this countermove that maximized the agents chance of winning. Upon introducing min and max nodes, our agent started to perform much better.

5 Future Improvements

A first improvement of note that we could make if time allowed would be to add a good evaluation function for our simulations. This would prevent the need to run simulations until the very end, instead allowing us to cut-off the simulations when we have a good idea that one player has a significant advantage, or if the result is likely to be a draw. Then, our time per simulation would be shorter and we would be able to run more simulations per move. Secondly, we believe that it would be advantageous to simulate more intelligent games. Given that the games we simulate are entirely random, the data we receive from simulations

from a specific game state is not entirely accurate, and in fact, it becomes increasingly inaccurate with less simulations. If for each move in our simulation, we were able to select a decent move in reasonable time, then perhaps the result we get from simulations would be more accurate, and our data on how good a given move is could improve. However, we are uncertain how feasible this would be in practice. It seems true that if were able to run the same amount of intelligent simulations as random simulations per move, clearly the intelligent simulations would yield better data. With that being said, it seems unlikely that we would be able to compute intelligent moves in the same time it would take to compute a random one. So there is a trade-off in terms of how intelligent our simulations are versus how many of these simulations we can make per move. Admittedly, this idea may not be very fruitful, but if time allowed, we believe it would be worthwhile to empirically observe how creating more intelligent and realistic simulations improves or worsens our agents play as opposed to entirely random simulations.

References

[1] Ben Ruijl, 2013, *Combining Simulated Annealing and Monte Carlo Tree Search for Expression Simplification*

[2] Artificial Intelligence: *A Modern Approach Textbook by Peter Norvig and Stuart J. Russell*