COMP 424: Final Project Report

Modified Monte Carlo Tree Search

Kais Jessa

April 13, 2021

1 Introduction and Motivation

My algorithm for this project is a combination of Monte Carlo Tree Search (MCTS) and a brute force depth-2 search. Due to the time limits on each turn in the tournament, the main criteria I considered when choosing an algorithm was that it should be computationally efficient and it should return the best move found thus far when interrupted, as this ensures that decent moves are always played without exceeding the time limit. In this regard, MCTS was the perfect candidate as performing random rollouts is quick and computationally inexpensive, MCTS converges to optimal play in the long run, and at any time, it can be terminated and a candidate move can be returned.[2] However, due to the randomized nature of MCTS, it may fail to find a win (or prevent a loss) in critical positions, especially in a short time-frame. For this reason, a simple depth-2 brute force search is applied before MCTS to ensure these critical moves are never missed. As will be shown below, the combination of MCTS with depth-2 search proves to be a very strong pentago-twist player.

2 Details of Implementation

2 Depth-2 Search

The depth-2 search is a simple brute force algorithm that looks for a "critical" move: a move that either ensures we win this ply, or blocks our opponent from winning on their next ply. Given the current state, the algorithm first generates all legal moves, then applies them to a clone of the current state and checks if we have a winning position. If so, we return this move and win the game. Otherwise, for each of these new states, we generate all possible moves for our opponent, apply these moves to clones of the new states, and determine whether they have a winning position. We then choose a move that does not lead to a winning position for the opponent (if such a move exists), thereby preventing them from winning on their next turn. If no critical moves are found after performing the search, we proceed to MCTS.

2 Monte Carlo Tree Search

To create the game tree, I wrote a Node class that stores information such as the board state, the move it represents from the previous state, the number of simulations played and the number of wins. Using the Tree class I wrote, a tree is initialized with its root node representing the current game state, and values of 0 for number of simulations and number of wins. We then proceed with the four-step process used in Monte Carlo Tree Search:

1. *Selection*: Starting at the root node, we repeatedly select the child node with the highest tree policy value until we reach a leaf node, that is, a node in the tree that does not yet have any child nodes. For the tree policy, I used Upper Confidence Trees (UCT) which balances exploration (choosing nodes with few simulations done thus far) with exploitation (choosing nodes that had high win-rates in previous iterations). The formula I used for UCT is $\frac{w}{n} + c\sqrt{\frac{\ln N}{n}}$, where n is the number of simulations the node has done thus far, w is the number of wins the node has thus far, N is the number of simulations the node's parent has done thus far, and $c = \sqrt{2}$ is a scaling constant.

2. *Expansion*: After selecting a leaf node, we create a new child node for every possible move from the current node's board state. We then arbitrarily choose one of these child nodes for the simulation. In practice, I simply selected the first node in the ArrayList of child nodes.

3. *Simulation*: Starting from the node's board state, we repeatedly select random moves until the game is over. We then return 2 if we win, 1 if we draw, and 0 if we lose. I previously tried choosing moves by performing a depth-2 search instead of at random so that the simulations are closer to real play, but this was inefficient and resulted in trees with very few nodes and simulations performed.

4. *Backpropagation*: Starting from the current node until we reach the root, we increment the number of games played by 2, the number of wins by the value returned by the simulation, and set the current node to its parent to move up the tree.

This entire process is performed in a loop that terminates after 1 second as this time limit gave me the most success with no timeouts (since the depth-2 search also uses some time). After looping, we select the child of the root node with the most simulations performed, since having more simulations means it was repeatedly a strong candidate in the selection phase and the algorithm prefers it over other actions.[2] We then return the move to the StudentPlayer which plays it on the board.

3 Evaluation of the Model

3 Advantages

One of the main advantages of Monte Carlo Tree Search is the ability to terminate the algorithm at any time and output a reasonable move. This allows the algorithm to generate good moves in any specified amount of time, whereas other search algorithms may need to terminate before finding the best move, leading to a random move being played in the tournament if the required search time exceeds 2 seconds. Another advantage is that MCTS does not need an evaluation function and instead determines the quality of moves through random playouts. Even with alpha-beta pruning and other optimizations, performing minimax without an evaluation function would require exploring a significant portion of the game tree as we would need to reach terminal board states, which is extremely computationally expensive and would not fit the requirements of the tournament. Although using an evaluation function would make minimax feasible, designing a strong evaluation function for the game could be difficult. Furthermore, minimax assumes that the opponent is also playing optimally with respect to the same evaluation function, meaning minimax could perform poorly against another player using a different evaluation function.[1] Therefore, not requiring an evaluation function is a significant advantage of MCTS.

3 Disadvantages

One of the primary disadvantages of MCTS is it determines the quality of moves through random playouts, which do not guarantee their optimality. For example, what would normally be considered a winning position could lose repeatedly during a simulation phase, meaning the algorithm will favour inferior moves. Similarly, nodes that consider losing moves could win through the random playouts, leading the algorithm to prefer it over superior moves. Although these incidents are rare, they could occur in practice, especially in short time frames where the algorithm is forced to return a move before it was able to fully consider these positions. However, performing a depth-2 search to find critical moves before performing MCTS can help mitigate this disadvantage, as it is guaranteed to either win in this ply or prevent a loss in the next ply if such a move exists. Although this does not guarantee wins or prevent losses over several moves, this small addition can prevent short-term losses in case MCTS does not consider them.

4 Alternative Approaches

I initially considered using only the depth-2 search (which returns a random move if no critical move exists) or only MCTS as both of these algorithms won 100/100 games against a random agent. To determine which model had the best performance, I had them play 100 games against each other, which yielded the following results:

MCTS vs. Depth-2: 79 wins, 20 losses, 1 draw
MCTS+Depth-2 vs. Depth-2: 77 wins, 12 losses, 11 draws
MCTS+Depth-2 vs. MCTS: 62 wins, 37 losses, 1 draw

As shown above, combining MCTS with depth-2 search generally had the best performance over using the algorithms individually, which is why I decided to combine these algorithms for my agent.

5 Future Improvements

When we expand nodes in the game tree, we create a child node for every legal move, including redundant and clearly suboptimal moves. For example, flipping or rotating empty quadrants all have the same effect but they are considered separate moves. One improvement would be to eliminate these moves outright before adding them to the tree, making the search more efficient as there are less nodes to consider. Also, the UCT formula used in the algorithm has a scaling constant of $\sqrt{2}$, but perhaps by comparing the performance of MCTS with different scaling constants, a more optimal value can be found, leading to improved performance and a better balance between exploration and exploitation.

One significant improvement would be to implement the Rapid Action-Value Estimation (RAVE) heuristic in MCTS, which allows knowledge to be shared among nodes that consider the same moves. This would lead to better estimations with less simulations, meaning we may even be able to remove the initial depth-2 search and allocate more time to MCTS.[2] Another improvement would be to replace MCTS altogether with some form of Iterative Deepening Search on the game tree where we perform minimax up to the first n layers of the tree (using an evaluation function on non-terminal nodes) and then incrementing n after each iteration. This maintains the benefit of storing the best move found so far but removes the randomized aspect of MCTS, meaning we would no longer require the initial depth-2 search. Iterative Deepening Search with a well-designed evaluation function, as well as optimizations such as alpha-beta pruning could lead to a strong agent overall.

References

[1] "Minimax." Wikipedia. Wikimedia Foundation. Accessed April 13, 2021. https://en.wikipedia.org/wiki/Minimax.

[2] "Monte Carlo Tree Search." Wikipedia. Wikimedia Foundation. Accessed April 13, 2021. https://en.wikipedia.org/wiki/Monte_Carlo_tree_search.