McGill University

Project: Pentago-Twist

COMP 424: Artificial Intelligence

Felix Simard (260865674)

Apr 13, 2021

Design and Motivation

To start off, I must say it was a true challenge, but also a great pleasure to build an agent capable of playing our relatively complex game of Pentago-Twist. Many very interesting design approaches covered during class lecture would've been suitable to tackle this project, whether it be the minimax algorithm, alpha-beta pruning, Monte Carlo Tree Search, Q-learning, supervised search, and more. From those, I must admit I've always been fascinated by the infamous Monte Carlo Tree Search algorithm for its clever utilization of randomness and impressive performance for large branching factor games. Throughout this report, I will discuss the overall design of my AI agent, my motivations, the theoretical basis for my approach, the main advantages and disadvantages, some other approaches I've explored, and conclude by presenting potential further improvements for my design. First, let's dive deeper into the design of my algorithm. Each time my agent's *chooseMove* method is called, three distinct steps are performed in the following order to determine the optimal move to play:

i. Scan all possible moves from the current state and check if a "win-on-next-move" exists which would allow my agent to win directly (*WinNextHeuristic*).
ii. If no "win-on-next moves" are found, check instead if our opponent could potentially beat us on its next move. If so, block the opponent by playing that critical move to try and avoid a loss. (*LoseNextHeuristic*).
iii. If no "lose-on-next moves" are found above, then run the Monte-Carlo Tree Search algorithm (MCTS), passing it the current board state (*MCTSExecuter*).

Overall, the first and second steps are somewhat trivial if none of the simple heuristic get triggered, meaning the core computation for a general move relies on our third step with the MCTS. To design and implement the MCTS algorithm, I relied mainly on Professor Cheung's lecture 10 slides [1], on the tutorial "Monte Carlo Tree Search for Tic-Tac-Toe Game" from the *Baeldung* website [3], as well as on some lecture slides "Extending MCTS" from the Computer Science Department of Swarthmore University [4]. Essentially, here is how my MCTS algorithm works:

While the time elapsed to find a move is still within the allowed limits, perform:

1. *Selection*
   Determine the most promising search tree node (*MCTSNode*) to expand based on the Upper Confidence Tree (UCT) policy by recursively computing the upper confidence value of the "best" node until a leaf node is reached in our search tree.

2. *Expansion*

   If the most promising node from step 1 is not a terminal state (i.e.: a win or loss), add all states (*MCTSState*) to the list of this node's children resulting from playing each move available at the current node state.
3. *Rollout/Simulation*

   For the simulation phase, if the expanded node from step 1 has any children, we simply select a random one of them and perform the rollout, if it has no children, then just rollout that node itself. Effectively, the rollout performs one single game simulation consisting of random moves until a winner is found.
4. *Backpropagation*

   Given the outcome of the previous rollout phase, we climb back up the tree from the terminal state reached in step 3 and update the *score* and *visits* value for all the nodes involved in the simulation that led to the win.

Finally, when the allocated time runs out, our agent breaks from the main loop and returns the optimal child from the node selected in step 1 based on the highest visits count.

Note that within the *MCTSState* class, I have also implemented a simple "states trimming" method which discards moves leading to duplicate states. This reduction is actually considerable as it allows our agent to reach greater depths of exploration quicker. Further explanation on this will be covered in later sections.

Hence, we have explained the design for three main strategies employed by our AI agent to choose its next move to play. In addition to being extremely curious about putting the MCTS algorithm to the test and truly understanding the almost magical result the algorithm gives, other motivations for using this approach include the convenience of implementing an algorithm which is not tied to some specifically defined heuristic of this particular game. Also, the MCTS algorithm is actually very straightforward to implement and does not require any complex or confusing programming logic. In the next section, I will present my understanding and research regarding the theoretical basis behind the MCTS algorithm.

Theoretical Basis for Approach

As covered in Professor Cheung's "Monte Carlo Tree Search" lecture [1], the idea behind the MCTS algorithm is to exploit random simulations to help us determine what move to select. In contrast to other popular approaches like Minimax, MCTS does not rely on having a specific evaluation function and does not assume that "both you and your opponent are playing optimally with respect to the same evaluation function" [1]. In fact, as shown through the design of my algorithm in the previous section, we saw that MCTS relies on four key steps: Selection, Expansion, Simulation, Backpropagation. In other words, at each iteration available for MCTS to perform, it will expand a certain node selected using a *tree policy*, and from the set of expanded states added to the search tree, a *default policy* will be used to simulate a chosen child node to then backpropagate the outcome of this simulation up the tree, updating visits and wins count for the nodes involved in the rollout. Now, as mentioned, MCTS does not rely on a game specific evaluation function, but rather on two policies: a *tree policy* and a *default policy*. For my implementation, which was inspired by the works shown in the tutorial "Monte Carlo Tree Search for Tic-Tac-Toe in Java" [3], the selection phase uses an Upper Confidence Tree policy (UCT) defined as follows:

$$Q(s,a) = \frac{node\ score}{node\ visits} + c\sqrt{\frac{\log\left(total\ visits\ of\ node\ in\ simulations\right)}{node\ visits}}$$

$where\ Q(s,a)\ is\ the\ value\ of\ taking\ action\ a\ in\ state\ s$ and c = $\sqrt{2}$

Effectively, the goal is to select, expand, and simulate the "most promising" node at each iteration to be as efficient as possible in finding the optimal move to play. Furthermore, the two terms involved in the tree policy equation aim to balance out exploitation (first term) and exploration (second term) with a given scaling constant c usually equal to $\sqrt{2}$. Subsequently, once a node is selected, we apply our *default policy* to choose an expanded child node for simulation. For the sake of simplicity and minimal cost, our implementation chooses a random child to initiate the rollout from and performs random moves throughout the simulation. Finally, the MCTS backpropagation step takes care of reflecting the outcome of each rollout by updating the visits count for all nodes in the simulation and the score counts for each of the nodes involved in a win.

Advantages and Disadvantages

In terms of pros, my algorithm implements simple “win-next” and “lose-next” heuristics which come in very handy to quickly return a pressing obvious move to play in the case where a winning move is available or when we have the opportunity to potentially block our opponent from winning. In addition, to avoid a large number of duplicate board states in our search tree, I implemented a *trimLegalMoves* method which reduces the branching factor at each node, hence, increasing the number of promising children the algorithm can explore within the allowed time. In fact, the number of duplicate states ignored at each iteration ranges from 0 to 175 depending on the depth of the node (closer to terminal state, less duplicate states possible). Now, for MCTS, the main advantages are, first, that it is an “anytime” algorithm, meaning it can terminate at any iteration and return the “current best” move to play. Also, as mentioned in the previous section, MCTS does not require any game specific rules, it is lightweight in comparison to other algorithms like Minimax and is suitable for games with large branching factors like our Pentago-Twist game.

On the flip side, my approach does present some disadvantages and weaknesses. In fact, even though this was stated as a pro, it can also be seen as a con to have no game specific heuristics employed by our algorithm which effectively prevents our agent from making more “informed” moves. Also, the time limitation at each move combined to the relatively large branching factor causes the exploration and expansion of the search tree to be somewhat suboptimal. Finally, looking at the UCT computation in my *MCTSExecuter* class, we see that our algorithm will eventually try to expand each child of our selected “most promising” node which adds some extra limitations on the exploration and exploitation our agent can perform within the time limits, which is also why trimming duplicate child states from our search tree was an important matter.

Other Approaches and Results

At a very high-level, I explored various implementations of the Minimax algorithm along with the alpha-beta pruning optimization, but I got quite confused when it came down to constructing a good evaluation function, so I decided to put this approach to the side. Now, as it will be reflected in the upcoming section, I wanted to have some mechanism for transferring the “knowledge” of each search tree constructed over to the next move in order to establish some kind of real “learning”. As you will observe in my code (commented out, but kept for completeness), I tried using Java’s serialization module to record the visits and score counts of nodes as well as the list of their children.

By saving such information inside a *.ser* file within the *data* directory, I attempted to see if that would have any success at making my agent learn across moves and games. Unfortunately, I was unable to converge to a working solution with that approach and I saw many potential flaws with it regarding overfitting certain moves or building up a way too large data file over time.

In terms of results for my current implementation, my agent beats the random player essentially all the time which is a good baseline. The table below testifies the performance of my MCTS agent (StudentPlayer) against the random player for 3 trials of 10 games each:

Trial	StudentPlayer Wins	RandomPlayer Wins	Draws
1	10	0	0
2	9	1	0
3	10	0	0

Overall, I expect my agent to perform relatively fine in the tournament, but I do foresee some performance lacking on my agent's behalf. But then again, as mentioned at very beginning, I am extremely satisfied and content of having played around with the MCTS algorithm and getting to really understand how it works in practice.

Further Improvements

As touched upon in the above section, having a learning factor transferred across moves and games which could keep track of node states, parents, scores, and visits count would greatly optimize the performance of my agent. For now, as seen in the main *chooseMove* method, I have to initialize a new *MCTSExecuter* instance each time a move needs to be determined which is quite inefficient given the short time allowed between moves. Hence, having some form of persistent knowledge or reinforcement learning strategy across games would have a huge impact on my agent's performance against any player. Furthermore, implementing a game specific heuristic to orient the choice of the child being simulated instead of picking one at random could very likely increase the chances of returning an even better move to play. Lastly, in terms of the data structures, constructs, loops, and more used to design the agent, it would definitely be valuable to optimize the actual code execution of my algorithm which would have a direct impact on MCTS's ability to explore and simulate more states resulting in an overall better move returned.

References

[1] J. C. K. Cheung, “Monte Carlo Tree Search,” in COMP 424: Artificial Intelligence, Lecture 10, 18-Mar-2021.

[2] G. Chaslot, M. Winands, J. Uiterwijk, and H. J. Van Den Herik, “Progressive Strategies for Monte-Carlo Tree Search,” *Research Gate*, Nov-2008. [Online]. Available: https://www.researchgate.net/publication/23751563_Progressive_Strategies_for_Monte-Carlo_Tree_Search. [Accessed: Mar-2021].

[3] Baeldung, “Monte Carlo Tree Search for Tic-Tac-Toe Game,” Baeldung, 03-Oct-2020. [Online]. Available: https://www.baeldung.com/java-monte-carlo-tree-search. [Accessed: 07-Apr-2021].

[4] “Extending MCTS,” CS Swarthmore University, 17-Feb-2016. [Online]. Available: https://www.cs.swarthmore.edu/~bryce/cs63/s16/slides/2-17_extending_mcts.pdf. [Accessed: 07-Apr-2021].

[5] “Serialization and Deserialization in Java with Example,” GeeksforGeeks, 10-Oct-2019. [Online]. Available: https://www.geeksforgeeks.org/serialization-in-java/. [Accessed: 07-Apr-2021].