Pentago-Twist - AI Agent

By Malena Mendilaharzu
260847841

Introduction

Game theory is a branch of mathematics used to model the strategic interaction between different players in a context with predefined rules and outcomes (Ippolito, 2019). In this report, we analyze the implementation of an AI agent designed to compete in a Pentago-Twist Game.

Design and Motivation

To compete in the Pentago-Twist Game, I decided to implement an AI agent that would use the Monte Carlo Tree Search as its main algorithm to come up with the following moves. In the beginning, I was unsure which one between Alpha-Beta Pruning (AB) and Monte Carlo Tree Search (MCTS) was the best search algorithm for this problem. However, after analyzing the game, I decided that MCTS was a better fit as the computational resources were limited by the instructions of the assignment and hence, simulation would be useful. Further, for AB Pruning to perform well against other's AI agents, I would have had to come up with an evaluation function and this would have needed a lot of knowledge of the rules of the game.

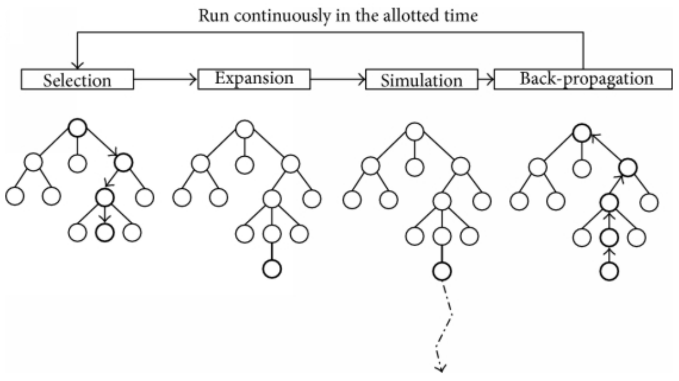
Therefore, the main design decision was to implement an AI agent based on the Monte Carlo Tree Search algorithm. In fact, this algorithm was proved to perform successfully in the so-called perfect information games. In short, perfect information games are games in which, at any point in the game, each player has perfect knowledge about all event actions that have previously taken place (Wyszyński, 2017). The Pentago-Twist Game is an example of a perfect information game as, throughout the game, all moves are visible to both players.

All perfect information games can be represented in the form of a tree data structure. The root of the tree would represent the beginning state of the game. For the Pentago-Twist game this would be the empty board. Then the children of each node would represent the following state after applying one of the possible legal moves that can be performed. Such illustration of the game is recursive, and the tree expands with a branching factor equal to the number of legal moves that can be done from each node. Such tree can then be used to decide which move is optimal.

The above-mentioned visualization of perfect information games was the main motivation behind the solution I provided to the Pentago-Twist problem. I used the Monte Carlo Tree Search algorithm to operate and handle the tree by simultaneously complying to the time and storage limitations. The 30 seconds allowed for the first move were used to expand this tree as much as possible so that more simulations could be done and therefore, better results could be achieved.

Theoretical bases of Monte Carlo Tree Search

The Monte Carlo Tree Search algorithm was introduced in 2006 by Rémi Coulom as a building block of the Go playing engine. As it names suggests, the algorithm simulates the games many times and attempts to predict the most promising move based on the simulation results.
To achieve this, four fundamental steps are needed: Selection, Expansion, Simulation and Back-propagation.



I. Selection

The selection step is called on a node (in our case it is called on the current root of the tree) and returns its most promising child by comparing their UCT values (Upper Confidence Bound for Trees).
The UCT value is calculated as follows:

$$\text{UCT} = \frac{w_i}{s_i} + c\sqrt{\frac{\ln s_p}{s_i}}$$

where:

- w_i represents the number of wins after the i-th move
- s_i represents the number of simulations after the i-th move.
- c represents the exploitation parameter ($\sqrt{2}$ was used)
- s_p represents the total number of simulations of the parent node

The child with the highest uct value is returned and considered as the most promising node. The idea is to keep selecting optimal child nodes until a leaf is reached.

II. Expansion

The expansion is based on the most promising node and consists in applying all legal moves to the state of that node: this creates all possible states that can be reached from the promising node. Then, these states get stored in new nodes that get added to list of children of the promising node.

III. Simulation

The simulation is based on a random child of the most promising node and consists in the acting of a game play – a sequence of moves that starts in the current node and ends in a node where a game result can be computed. In this step, moves are chosen with respect to a rollout policy function. In practice, we want this policy to be designed as fast as possible to allow for many simulations. A commonly used policy is a uniform random function. Simulation always results in an evaluation – whether it was a win, a loss or a tie. In the code, the score of the node is set to the Integer.MIN_VALUE if our opponent wins and to the Integer.MAX_VALUE if we win. No node update is done for ties.

IV. Back-Propagation

The final step of the Monte Carlo Tree Search is back-propagation, and it consists in using the result of the simulation to update the score values of all nodes previously selected. Back-propagation also updates the visit number of each node.

Using Monte Carlo Tree Search to select the next move:

The above-mentioned four steps are used to select the returned move. The idea is that we run those steps for the amount of time given (30 seconds if it's the first move, 2s for any other consecutive move) and then when the time is up, we select the move of the child of the current node with the highest score. We estimate that doing this will return a move that will most likely lead us to win the game.

Advantages of this approach

The main advantage of using Monte Carlo Tree Search is that it performs well with high branching factors. The Pentago-Twist Game is a game with many possible moves which makes the tree grow exponentially. Random expansion and simulation of promising nodes allows to overcome the issue of having to check every single node in the tree. Another advantage of this approach is that no matter how much time is given to perform the computations, the algorithm always builds up its solution towards a better result. This means that it will always return a move with higher chances to lead to a win than a random move. This is particularly good for the time limitations we have: Monte Carlo Tree Search will focus on the promising nodes and return the optimal it has found so far. Finally, one last advantage of this technique is that it doesn't require the implementer to be an expert in the game as the algorithm uses no heuristic to determine the next move.

Disadvantages of this approach

One disadvantage of using Monte Carlo Tree Search as our searching algorithm is that, unlike Alpha Beta Pruning, it does not guarantee the optimal move. Due to the randomization of the simulation, it is possible that non-optimal moves are selected and preferred over others. Further, not having a heuristic can also be seen as a disadvantage as it might leads us to end up using more time resources to compute the next move.

Alternative approaches

As previously mentioned, another approach that was considered to solve this problem was to use Alpha Beta Pruning as the search algorithm to find the move. As many researchers found, with a good evaluation function, Alpha Beta Pruning can actually outperform MCTS and even take less time to perform the computations. Therefore, my initial approach was to attempt to design a good evaluation function. The basic idea was to, for each tile check its surrounding lines (horizontal, vertical, left diagonal and right diagonal) and count how many tiles of the same color we had. The score added to the node would then be proportional to the number of tiles in a line and inversely related to the same number for the tiles of the opposite player. However, I then decided to abandon this perspective as I preferred the abstraction from the rules of the game that Monte Carlo Tree Search allow me for.

Another approach I took was to make the Monte Carlo Tree Search more exploratory or exploitative by changing the exploitation scalar c. I thought it would have been a good idea to change this scalar throughout the execution of the program. However, no improvement was seen so I decided to use the scalar factor proposed in class.

Future Improvements

Some future improvements for my agent would be to store the data from past games in a file and use this data to avoid performing computations that we already encountered. We could use parallel programming to search through the file and run our search algorithm simultaneously. I would also like to try different UCT evaluation functions such as RAVE or PUCT and see how they affect the performance of the algorithm. Finally, ideally, I would want to apply Machine Learning to my agent: perhaps I could have my agent to learn from the moves of the best real human Pentago-Twist players.

Sources:

Ippolito P. P. (2019). Game Theory in Artificial Intelligence. Towards Data Science https://towardsdatascience.com/game-theory-in-artificial-intelligence-57a7937e1b88

Wyszyński M. (2017). An Introduction To Monte Carlo Tree Search. Appsilon Data Science https://www.kdnuggets.com/2017/12/introduction-monte-carlo-tree-search.htm