Monte Carlo Tree Search Algorithm Applied to Pentago Twist

1 - Motivations & Decision-Making Logic

Pentago Twist is played on a 6x6 board composed of 4 quadrants with each quadrant being a 3x3 board. To win the game, a player must successfully place 5 of their pieces consecutively (i.e. in the same row/column/diagonally). Further, after placing their piece, the player must select one of the four quadrants and either rotate that quadrant 90 degrees right or horizontally flip that quadrant.

The extended board size and the rotating/flipping feature greatly increases the number of possible moves for any given turn when compared to similar games like Tic-Tac-Toe. For example, on the first turn, a player has 36 possible choices to place their piece & for each possible choice they can select one of the four quadrants and decide to either flip/rotate that quadrant. This large amount of possible moves to make per turn is the reason that Pentago Twist is a much more complex game & has a **very high branching factor**. Branching factor is the # of branches that a node might have in a search tree.

Due to having such a **high branching factor**, it makes the classic search algorithms not plausible (due to time & memory constraints) since attempting to create a search tree from this game will be gigantic since one can imagine that each move can have upwards of 100 choices, so creating a tree with all possible moves and searching through the many nodes wouldn't even be possible. This is the primary motivation for using the **Monte Carlo Tree Search** (MCTS) algorithm for deciding next moves on the fly during the game since the MCTS algorithm excels for scenarios with high branching factors (explained in the theoretical section below).

The secondary motivation for applying the MCTS algorithm here is **making decisions require no knowledge of specific game strategies** of any sort. This is because the algorithm doesn't use any kind of heuristic function, so there is no assumption being made about how the opponent may be optimizing their game strategy.

The MCTS algorithm was selected for precisely these two reasons.

The program's logic for choosing a move (chooseMove method) is broken down as follows:

1) It looks at the current board and iterates through all possible legal moves that the bot can make. For each move, it checks if the resulting board after processing that move results in a win for itself. If such a move exists, we immediately return this move. This is basic logic that guarantees to choose the winning move if one exists. (**IMMEDIATE WIN)**
2) It then processes 2 different possible moves on cloned boards & then generates all the possible moves an opponent can make after those moves & it checks if the opponent has any possible way of winning on their next turn. If there is a move they can make to win, then the bot will play this move to block the opponent from taking that spot on the board provided the move is legal to make. **(BLOCK OPPONENT)**
3) If there is no immediate winning move & there's no need to block an opponent's immediate winning move, then the bot resorts to the MCTS algorithm for deciding where to play on their turn. This is the provider of the vast majority of move decisions for this bot. **(MCTS DECISION)**

2 - Theory Behind The MCTS Algorithm

In the following context, a tree is a data structure containing nodes & the root of the tree is the only node in the tree without a parent. A node in the tree is a representation of a specific board state (a specific setup of the game pieces). All the children of a node are new specific board states that are generated by applying a valid move from the state of the original node.

The Monte Carlo Tree Search algorithm is made of 4-steps:

1) **Selection** – uses the **tree policy** to select a path of nodes in the tree. This tree policy is the Upper Confidence Bound applied to Trees (UCT) formula to assign a UCT value to every node & selects the node with the max UCT value until it reaches a leaf node.
   The UCT formula is:

$$Q^{+}(s,a) = Q(s,a) + \sqrt{2 * \frac{\log n(s)}{\log n(s,a)}}$$

where

Q(s, a) is the value of taking action a in state s (wins / total visits)

n(s, a) is the number of times we have taken action a in state s

n(s) is the number of times we have visited s in simulations

(from class notes)

2) **Expansion** – expands the tree (generates all children nodes) of the selected leaf node from step 1. This is done by getting a list of all the valid moves of the game state of the selected node & applies every possible move, creates a new node & appends that node to the list of children of the selected node.
3) **Simulation** – Randomly selects one of the expanded nodes from step 3 & randomly playout the rest of that game (by making random move decisions for both players until there is a win / loss / draw). This decision making for playouts is called the **default policy** (cheap / computationally inexpensive).
4) **Backpropagation** – at the end of the simulation from step 3, the result win / loss / draw value is sent upward and updates the visit counts for each node visited in that path.

This is the exact algorithm implemented in the game-playing bot. It runs these 4 steps over and over for as long as it can until the two second timer runs out to make a decision, as per tournament constraints. At the end of iteration, the child of the root (current game state) with the max win rate value is returned to the server and is the MCTS move decision.

This shows that no game strategy is needed as it uses properties of probability & statistics with the use of the UCT to optimally balance exploring different branches with exploiting a promising branch. Through random simulations, as the sample size (i.e. # of iterations) approaches infinity, the MCTS decision would tend to the optimal decision.

It excels in games with high branching factors because of the property of it not being affected by branching factor since the search can concentrate on promising paths much more and go deep in that path.

3 – Advantages / Disadvantages

The advantages of using the MCTS algorithm for decision making in Pentago Twist are:

1) It's able to provide a meaningful decision within the two second constraint for a game with a huge branching factor, as mentioned in the first section.
2) It makes no assumption of what the opponent is optimizing for like in the minimax algorithm / Alpha-beta algorithms which require an evaluation function / heuristic function which evaluates how "good" a move is for the player and its opponent with the same function. This also assumes the opponent is behaving optimally according to the evaluation function, which is rarely true in the real world. This allows for competition with any general opponent with any general strategy and will not crumble versus opponents whose strategy differs.

The disadvantages of using the MCTS algorithm are:

1) Due to its random nature of random playouts, the simulation can be unlucky and have bad simulations for a good move, which would cause the algorithm to not exploit that branch much further. This is why the algorithm will often choose suboptimal moves since we are not able to run an infinite number of simulations.

5 – Improving the Player

Some future improvements to the player would include:

1) Changing the default policy from randomly simulating the game decisions to using some kind of informed search algorithm such as alpha-beta, but this would allow for less iterations of the MCTS algorithm since running Alpha-beta is more computationally expensive than a simple random simulation. It's a trade of cost for better accuracy of a given node.
2) Use of machine learning to learn good heuristic functions over a large sample set of games & use this heuristic function for the default policy as mentioned in (1) above.
3) Through a large, labeled sample set of completed matches with numbered opening moves, use learning, once again, to adopt an optimal opening strategy for the player instead of simply running MCTS on a blank board (not enough time to generate an optimal move).