# Samtago-Twist:
# A Monte Carlo Tree Search Agent for Pentago-Twist

Samuel Morris, 260833487

COMP 424, Introduction to Artificial Intelligence

Professor Jackie Cheung

April 11, 2021

## Motivation for Design

The *Samtago-Twist* agent is based off of the Monte Carlo Tree Search (MCTS) algorithm. My motivation for using this approach is twofold. First, MCTS "can … handle large search spaces with high branching factors" (Baier & Winands, 2013). It is known that Pentago-Twist has a high branching factor: "there are 36 · 8 = 288 possible first moves including rotation and an average branching factor of 97.3 over all states" (Irving, 2014). So, for this reason, MCTS seemed like an appropriate approach. Second, MCTS does not require a handcrafted evaluation function (Baier & Winands, 2013), which is ideal because designing a good evaluation function requires prior knowledge of the game. And I have no prior knowledge of Pentago-Twist.

## Theoretical Basis of MCTS

MCTS works by selecting a leaf node, expanding it (i.e., generating its children), performing a rollout (i.e., playing the game according to some policy until it is finished), and propagating the results back up to the root of the tree (i.e., keeping a live tally of games won vs. games played for each node) (James et al., n.d.). Two policies are needed to implement a MCTS: a tree policy and a rollout policy. The tree policy selects the next leaf node to perform a rollout, and the rollout policy performs a rollout by selecting moves for each player until the game is finished. *Samtago-Twist* uses the Upper Confidence Bound (UCB1) as its tree policy, which serves to balance exploration and exploitation when selecting a leaf node. UCB1 is the most popular approach to MCTS (Lieck et al., 2017). And this is my main reasoning for using it. *Samtago-Twist* uses a stochastic rollout policy because it is simple to code and runs quickly. I attempted to implement a heavy rollout policy, but found that performance worsened when compared to the stochastic policy (see notes on improvement for a description of a heavy rollout policy).

## Advantages and Disadvantages

Using a MCTS approach has many advantages. First, it can halt and return a solution at any time. This is particularly useful because it means I don't have to worry about the algorithm not terminating before time runs out. Second, as previously mentioned, MCTS is adept at handling large state-spaces. This means that it can still return a fairly good move even when the branching factor is extremely large. Third, and also previously mentioned, is the fact that MCTS does not require an evaluation function. This allows me to program a successful agent without in-depth knowledge of how Pentago-Twist works, something that designing an evaluation function usually requires.

MCTS also has disadvantages. First, it does not learn from its mistakes. Because MCTS is not a machine learning algorithm, it lacks the capacity to learn and does not use a learned policy to select its moves. Second, its rollout policy can affect performance. By simulating rollouts randomly, MCTS risks getting unlucky and performing rollouts whose results do not accurately represent the value of the board; and by using heavy rollouts MCTS risks letting "an incorrectly biased policy … cause [it] to focus initially on a completely suboptimal region" (James et al., n.d.).

## Notes on Improvement

As much as we would like to believe that *Samtago-Twist* is perfect, it can inevitably be improved upon. Here, I present two possible ways that this can be done, both of which I attempted to implement but found that after testing could not consistently beat the standard MCTS player. The first strategy is to use heavy rollouts (rollouts that choose the next action by performing a single-move lookahead) to simulate games from the leaf nodes. Heavy rollouts can be better than stochastic ones because they provide more information about the value function if the state-space is very large (James et al., n.d.). This is beneficial because it initially focuses the search on parts of the game tree that are more promising. The second strategy is to use rapid value action estimation (RAVE) to more rapidly estimate the value of taking a specific action, regardless of when the action was played. This works by updating the number of wins in the siblings of the rollout node that match the actions played in the rollout, and assumes that move order is irrelevant in determining the value of an action (Gelly & Silver, 2011).

## References:

Choudhary, A. (2019, January 23). Monte Carlo Tree Search Tutorial | DeepMind AlphaGo. Analytics Vidhya. https://www.analyticsvidhya.com/blog/2019/01/monte-carlo-tree-search-introduction-algorithm-deepmind-alphago/

Baier, H., & Winands, M. H. M. (2013). Monte-Carlo Tree Search and minimax hybrids. 2013 IEEE Conference on Computational Inteligence in Games (CIG), 1–8. https://doi.org/10.1109/CIG.2013.6633630

Gelly, S., & Silver, D. (2011). Monte-Carlo tree search and rapid action value estimation in computer Go. Artificial Intelligence, 175(11), 1856–1875. https://doi.org/10.1016/j.artint.2011.03.007

Irving, G. (2014). Pentago is a First Player Win: Strongly Solving a
Game Using Parallel In-Core Retrograde Analysis. Otherlab
San Francisco, CA. https://arxiv.org/pdf/1404.0743.pdf

James, S., Konidaris, G., & Rosman, B. (n.d.). An Analysis of Monte Carlo Tree Search. 7.
http://cs.brown.edu/people/gdk/pubs/analysis_mcts.pdf

Lieck, R., Ngo, V., & Toussaint, M. (2017). *Exploiting Variance Information in Monte-Carlo Tree Search*. 9. https://ipvs.informatik.uni-stuttgart.de/mlr/papers/17-lieck-ICAPSws.pdf