Monte Carlo Tree Search Cannot be Beaten by Random Player in Pentago Swap

Final Project - COMP 424: Artificial Intelligence

Le Nhat Hung - 260793376

McGill University - April 5, 2019

1 Introduction

Pentago-Swap is a variation on the game Pentago. Two players play against each other on a 6×6 grid, divided into four 3×3 quadrants. A move consists of placing a piece and swapping any two quadrants (instead of rotating one quadrant like in regular Pentago). To win, a player needs to have 5 of their pieces in a row, whether it is horizontal, vertical or diagonal.

The proposed work is a AI Pentago-Swap playing agent based on the Monte Carlo Tree Search (MCTS) algorithm, using an Upper Confidence Bounds applied to Trees (UCT) function for tree traversal. The agent operates under certain restrictions: it has 30s to make the first move, 2s for all subsequent moves, and cannot use more than 500MB of RAM.

1.1 Preliminaries

MCTS [Chaslot et al. 2008] is a game-playing algorithm which represents the possible states of a game by a tree. It chooses the move promising move through a **search.** A search is a set of **traversals** down the tree. A single traversal is the path from the root node (current game state) to a node that is **not fully expanded.** A not fully expanded node has at least one of its children (set of possible next game states) is **unvisited.** When a not fully expanded node is reached, one of its unvisited children will be chosen: from its state, the game is **simulated** until a **terminal node** is reached where the player either won or lost. The result of the simulation (win or loss) is then **propagated back up** (backpropagation) to the tree root, updating the **utility** of traversed node. Once many simulations are completed, the root's child with the highest utility will be chosen as the actual move.

UCT [Kocsis and Szepesvári 2006] is the function which computes the utility of each node during back-propagation. MCTS without UCT simply defines a node's utility as its simulated win rate. However, UCT enables control of exploration over exploitation, which for example can prevent the tree search to avoid a node that had one "unlucky" simulation where the player lost.

2 Motivation

A domain-specific reason for choosing MCTS over Minimax or α-β pruning [Saffidine, Finnsson, and Buro 2012] is the resource restriction (time limit and RAM), as both of the alternatives require a full expansion of the search tree.

As for UCT, the control of the exploitation-exploration ratio it offers has been shown to be more effective than regular MCTS in various domains Kocsis and Szepesvári 2006.

The choice of MCTS and UCT is also motivated by the recent breakthrough that is AlphaGo Zero (AGZ) [Silver et al. 2017]. The paper by Google DeepMind presents an AI agent boasting superhuman performance in the game of Go, achieved from reinforcement learning only i.e. pure self-play - no training data were taken from matches against human players. AGZ also used MCTS and UCT, albeit coupled with a convolutional neural network (CNN) with residual layers, trained simultaneously as a match progresses. Another parallel is that Go falls into the Moku family of games, the same family as Pentago's.

3 Approach

AGZ is the reference model for the functions and parameters that went into creating this Pentago-Swap agent. The main difference is the lack of a neural network, which for AGZ produces a factor to be used in its UCT function, which will not be present here. It follows that no machine learning is involved.

Each node (s, a) in our search tree stores a game state s and the move/action a that led to s. The UCT function computing the utility of (s, a) is:

$$\mathrm{UCT}(s,a) = Q(s,a) + c\sqrt{\frac{N(\text{parent node of } (s,a))}{N(s,a)+1}} \tag{1}$$

where $Q(s, a)$ is the action value of the node (s, a), $N(\cdot)$ for any node is the total number of times it has been traversed, and c a constant scalar controlling the degree of exploration i.e. higher c means more exploration. It follows that the left term in UCT is the **exploitation component**, and the right term is the **exploration component.** Further, $Q(s, a)$ is computed from:

$$Q(s,a) = \frac{\sum_{s'} V(s')}{N(s,a)} \tag{2}$$

where the set of s' is the set of terminal nodes reached from simulating (s, a), and $V(s')$ is the value associated with the outcome of s': a positive reward value if the player won, or a negative penalty value if the player lost. For example, with a reward $= 1$, penalty $= 0$ and $c = 0$, $\mathrm{UCT}(s, a)$ would simply be the win rate of the node (s, a), which is analogous to the utility function of regular MCTS.

When it is the agent's turn to play, a new tree is created with the current game state as the root. It then freely performs simulations within the allotted time (25000ms and 1900ms instead of 30s and 2s to deal with overhead), then chooses the child of the root node with the highest utility computed from UCT and returns its action.

There are therefore 3 main hyperparameters to set:

1) c, which scales the exploration component of UCT and thereby controlling exploration,

2) the reward,

3) and the penalty.

In the submitted work, c is set to $\sqrt{2} = 1.4$ and {reward, penalty} to $\{100, -100\}$. This would unfortunately prove to be shortsighted, as these values were shown to be suboptimal in subsequent tests. Due to time constraints, however, these values are the one which ended up in the submitted code. Nonetheless, the results and findings all of conducted tests will be presented here.

3.1 Other Approaches

The submitted agent creates a new search tree at each move, in line with the standard implementation of MCTS. However, other works including AGZ maintains one persistent tree throughout one whole play-out. This was the first approach as it presented many desirable advantages. First, there would be no overhead of possibly reinitializing nodes that had already been traversed in the previous search tree. Second, it makes

first move time limit (ms)	1900	5000	7000	10000	25000
normal move time limit (ms)	1900	1900	1900	1900	1900
times out	no	no	yes	yes	yes
win rate (%)	33	33	0	0	0

Table 1: Persistent-tree agent vs. regular agent

intuitive sense that the UCT value of each node would be more accurate if it is maintained throughout a whole game due to the law of Large Numbers. However, testing showed that the persistent tree agent underperforms against a regular agent at various time limit settings (Table 1). With a longer time to make the first move, the persistent-tree agent can build a larger tree at the start of the game. However, the larger the tree, the more children there are to search for a matching state and action pair (s, a). The search would then exceed the time limit. Shortening the starting move time limit remedies this, but persistent-tree underperforms still. This is because later simulations would favor already explored nodes as they already possess higher UCT values from previous moves.

Next, testing was conducted to find the optimal value of c. The theoretically optimal value of c is $\sqrt{2} \approx 1.4$ [Kocsis and Szepesvári 2006]. However, to test its effectiveness in this specific setting, an exploitation only agent ($c = 0$) was pitted against an exploring agent with $c = 1.4$. Each agent played made the first move in alternating matches. Out of 100 matches, the exploitation-only agent won ≈ 70 of them. This result is admittedly rather disappointing, as it seems in this specific setting, pure exploitation is optimal, which defeats the purpose of implementing UCT.

Finally, the effect of setting a high penalty on performance was tested. Basic implementations of MCTS set the reward to 1 and the penalty either 0, or to -1 as with the case of AGZ. However, since this agent lacks the advantage of being trained through reinforcement learning, deciding how grave the penalty should be compared to the reward greatly influences the UCT function and might lead to significant performance improvement. For example, a relatively unpopular blog post from 2015 [Gijs-Jan 2015] found that with {reward, penalty} set to $\{1, -100\}$ gave MCTS a "very paranoid behavior," traversing much deeper down the tree through nodes "which gave the opponent as little as opportunity as possible." This seems to be untrue for this specific setting, as a $\{1, -100\}$ agent was tested against a $\{1, -1\}$ and only won 33% of matches. This result is both a bit underwhelming and reassuring, as the submitted agent has the reward and penalty set to $\{100, -100\}$, which is different from $\{1, -1\}$ in scale but identical in proportion. However, further tests showed the $\{1, -1\}$ agent does perform better, winning 60% of matches.

In summary, the submitted agent would benefit from having the c parameter set to 0, and the reward and penalty set to $\{1, -1\}$. Unfortunately, these changes would undo the effect of the UCT function. The end product from this change would be a regular MCTS algorithm with a negative penalty. The submitted agent differs from the "best" agent in that its c parameter is 1.4 instead of 0, and its reward and penalty are set to $\{100, -100\}$ instead of $\{1, -1\}$.

4 Pros and Cons

This implementation offers a few merits. First, it does not involve machine learning. This means the implementation is simple and there is no starting load time needed for file IO before making the first move. The lack of a neural network also makes the inner workings more interpretable, with considerably fewer hyperparameters to set. It achieves reasonably good performance despite its simplicity, and can act as a baseline for further improvements.

However, the implementation has many drawbacks. First, it does not involve machine learning. The agent does not have superhuman performance, as I have personally verified myself. It also did not make use of multithreading, which would greatly scale up the number of simulations performed before choosing a move.

5 Future Work

The finding that c parameter would have been better set to 0 is unfortunate, as there would no longer be a degree of exploration. Perhaps exploration would have been better handled if a neural network was trained simultaneously like with AGZ. A neural network implementation would definitely be worthwhile to dive into, as it would lead to deeper understanding of AGZ's inner workings and almost certainly better performance.

References

Chaslot, Guillaume et al. (2008). "Monte-Carlo Tree Search: A New Framework for Game AI". In: *AIIDE*.

Gijs-Jan (2015). *Walk the Line - MCTS Evaluation Functions*. URL: `codepoke.net`.

Kocsis, Levente and Csaba Szepesvári (2006). "Bandit Based Monte-Carlo Planning". In: *Machine Learning: ECML 2006*. Ed. by Johannes Fürnkranz, Tobias Scheffer, and Myra Spiliopoulou. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 282–293. ISBN: 978-3-540-46056-5.

Saffidine, Abdallah, Hilmar Finnsson, and Michael Buro (2012). *Alpha-Beta Pruning for Games with Simultaneous Moves*. URL: `https://www.aaai.org/ocs/index.php/AAAI/AAAI12/paper/view/5181`.

Silver, David et al. (2017). *Mastering the game of Go without human knowledge*. URL: `https://www.nature.com/articles/nature24270/`.