An AI Player for Pentago-Twist

Christian Zhao

January 5, 2022

1 Implementation

1.1 Overview

The working agent of the game Pentago-Twist decides what move to choose at each turn based on both the UCT (variation of Monte Carlo tree search) algorithm and the alpha-beta pruning algorithm: When playing as the first player (white), the agent uses UCT, and when playing as the second player (black), the agent uses alpha-beta pruning. There are five main components. First, a strategy selector located in *StudentPlayer.java*, which selects the strategy for move choosing. Second, a decision maker based on UCT, located in *MCTS.java*. Third, a decision maker based on alpha-beta pruning, *ABPruning.java*. Fourth, some useful helper methods and classes in *MyTools.java*. And last, a hand-crafted utility function *ImprovedEval.java*.

Open Moves

The strategy for the first two turns are fixed: the agent chooses an empty spot at the middle of any quadrant. This is simply because the middle spot is involved in highest number of lines that could lead to a win and are fixed. After the first two turns, the agent uses UCT or alpha-beta pruning.

UCT

The basic Monte Carlo tree search algorithm has four stages: selection, expansion, simulation, and back-propagation, and this is the skeleton of my implementation of UCT.

Selection In the selection stage, the agent chooses the most promising child of each node until a leaf is reached, using UCB1 with an exploration constant of 1.4 as the selection algorithm, combined with the utility function. So the value function for each child node is

$$Q^*(s,a) = Q(s,a) + c \cdot \sqrt{\frac{\ln n(s)}{n(s,a)}} + U(s,a),$$

where $c = 1.4$, U is the utility function. Another important design decision is that, when the number of visited times of a node is 0, its UCB value will be 1 instead of a very high number. Otherwise, since the branching factor is high and only thousands rollouts can be played in two seconds, the agent will often be distracted from going deeper of the tree, and thus favor exploration too much to exploitation.

Expansion In the expansion stage, the agent would decide to expand the leaf node if 1. the leaf node is not a terminal state. 2. the node has been visited. This is the standard condition for expansion of the leaf node. In expanding the leaf node, moves that lead to the same state are truncated, leaving only moves with distinct results. This is done via *mergeNextStatesPartial* in *MyTools.java*.

Simulation and Back-propagation In the simulation stage, the agent would simulate 20 games from the leaf state instead of 1 for a better approximation, at the cost of smaller total number of iterations - this is a trade-off between quality and quantity of simulations. If the leaf state is a terminal state, or the leaf node's direct child is a winning state for

the player at this leaf node, the numbers of wins and total games are immediately back-propagated after applying a lever constant of 5, because such a state that has a definite winner carries exact information of the quality the state, and thus is worth more than the simulation result of non-terminal game state. After simulation, the result is back-propagated to all nodes in the path from the root to the leaf, increasing the number of simulations and the number of wins for the root player.

Alpha-beta Pruning

Alpha-beta pruning is not only used for the second player. After the turn number exceeds 14, the agent automatically switch to using alpha-beta pruning, since the search tree of end-game can be thoroughly traversed. The standard alpha-beta pruning algorithm is implemented, with a transposition table storing results for visited states and elimination of moves with the same effect (a method in *MyTools* similar to the one used in the expansion phase of the UCT agent).

Utility Function

The alpha-beta pruning agent and the UCT agent use the same hand-crafted evaluation function. The evaluation function calculates the assigned value for each configuration of each winning line (18 in total) during the processing time of the first move, and stores these values in arrays. Calculating the utility of a state requires only gathering the state of each winning line, calculating the indices, retrieve values for each line from the arrays, and sum them up.

The values for each configuration is assigned based on intuition and are somewhat arbitrary, as there are too many parameters. This utility function could be better by understanding the game more deeply, though since the time needed for writing a second utility function is unaffordable, I did not explore further on this.

Please see the code for exact implementation details of each component.

1.2 Motivation

First of all, multiple runs of the game show that the alpha-beta pruning agent has a higher winning rate than the UCT, but after weighing the cons and pros, I decided to use UCT and alpha-beta pruning respectively for the first and second player.

The restriction on the agent is a major reason for this global strategy - for each move there are only two seconds allocated. In testing alpha-beta pruning, it appeared that the time take to choose a move is quite unstable for a depth-limit of 3 - almost in every properly played game (more than 15 pieces placed), there are at least 3 steps in mid-game that take the agent more than 2 seconds, and playing 3 random moves could be deadly. This is a direct result of the high branching factor of the game - the search tree with depth 3 in mid-game could have $\sim 200^3 = 8$ million leaf nodes.

Testing also shows that the agent has to search at least 3 plies (1 turn counted as 2 plies) to make a good move, and there is an obvious drop of the quality of the agent's decision if searching stops after reaching the time limit (shown in the Comparison section), which is not a surprise as alpha-beta pruning has no guarantee for the quality of the move chosen before the entire search tree is traversed.

Imposing time constraint on UCT is easy - I simply repeated the four stages until the time limit is reached. Unfortunately, the performance of UCT is worse quite a lot than alpha-beta pruning, and is not as stable as alpha-beta pruning due in that it uses the random policy for simulations.

The deciding factor for choosing this strategy is a property of the game: the first player has a higher possibility to win than the second player, as can be seen from the result table below of 1000 games of two random agents playing against each other.

	# wins as White	# wins as Black
random agent	249	214
random agent	250	210
# draws	78	

Given this fact, I decided that since the average performance of UCT is worse than the alpha-beta pruning while UCT has a more stable result than alpha-beta pruning which occasionally takes too long, using UCT as the first player can ensure a stably high possibility to win. When playing as the second player, the agent is given a lower bound of possibility to win, and using alpha-beta pruning to gamble on a possible win is more favorable than using UCT, which has a stable performance and thus unlikely to play well as the second player.

2 Theoretical Foundation

Alpha-beta pruning is an improved version of the minimax algorithm, which essentially chooses the best move in worst case (assuming an optimal opponent) by searching through all possible games from the current state. Alpha-beta pruning differs from simple minimax in that it prunes a subtree of the root immediately once it sees that the subtree is determined to yield a result worse than the current best result [1]. However, the worst case time complexity of alpha-beta pruning is the same as minimax, $O(b^n)$, where b is the number of legal moves at each state and n is the search depth.

UCT is based on naive Monte Carlo tree search with a greedy action selection. Greedy action selection could lead to suboptimal results, as it does not take into account the fact that a low winning rate of state with a few simulations could be highly inaccurate. UCT applies the UCB1 algorithm to Monte Carlo tree search for balance between exploitation and exploration, and is proven to converge to the minimax action as the number of simulations increases[2].

The logic of my hand-crafted utility function is inspired by the utility of a student's pentago AI [3], which counts the number of pieces on each winning line and assigns weight to each number, i.e.

$$U(s) = \sum_{i=1}^{18} w(n(l_i)),$$

where $n(l_i)$ is the number of the player's pieces on i-th winning line, and w is the weight function. Yet my evaluation function is quite a lot more sophisticated than Niklas' hoping to get a closer approximation to the true utility, as it considers more cases for each number of pieces, for example, a configuration of three consecutive pieces on the same quadrant is worth more than other configurations.

3 Advantages and Disadvantages

In this section, I will discuss the advantages and disadvantages of alpha-beta pruning and UCT, and specific problems of my implementation.

The most prominent advantage of alpha-beta pruning is its quality of move chosen given that the search tree is thoroughly traversed. Using the hand-crafted evaluation, the alpha-beta pruning agent as the first player is always able to beat me, while the UCT agent sometimes cannot. It is also simple to implement, as the pseudo-code of the algorithm can be easily implemented. However, its disadvantage is also obvious. Alpha-beta pruning requires searching

through the entire search tree to ensure an optimal result given a depth limit, and there is no guarantee how good the agent's decision could be when it is terminated. When there is a hard time restriction on the alpha-beta pruning agent, the performance drops quite a bit. And last, alpha-beta pruning is a deterministic algorithm. If the utility function is perfect, then the deterministic agent would also be perfect, but since my hand-crafted utility cannot be as good, it means that the agent will always perform the same bad moves resulted from the intrinsic inaccuracy of the utility function in the same state.

UCT, on the other hand, has the great advantage that the time it takes can be controlled precisely while not impacting the quality of decision making as significantly as in alpha-beta pruning - I can simply run rollouts repeatedly until the time limit is reached. However, since UCT keeps track of all nodes of the entire tree, its space complexity is much higher than that of simple alpha-beta pruning, which is on the order of the maximum depth of its search tree, though in my implementation, the transposition table of alpha-beta pruning would probably take as much space as UCT. Moreover, given the high branching factor of the game and time restriction of two seconds, the number of simulations one can run is far from sufficient for it to converge to the minimax action result.

Another disadvantage of UCT is its complexity for implementation, and this can be best seen by contrasting to alpha-beta pruning. There is not much one can do with the logic of the alpha-beta pruning algorithm. To improve the performance of alpha-beta pruning, one can at most try to obtain a better utility function, or optimize the time performance of the algorithm, for example, memorizing visited states. For UCT, there are many factors that influence the performance of the agent, including the design of the selection algorithm (one can, for example, add utility of the state to UCB1 as I did, or use a variant of UCB1), the way simulations are conducted (random move or a simple policy), and the result is that it requires a lot of tuning to make the agent work properly - this can be best seen in the complexity of the code itself.

In practice, when the UCT agent is applied to early-game as the second player, it sometimes fails to block the first player from getting a consecutive 3 pieces in one quadrant, and occasionally makes moves that seem no better than random moves. Alpha-beta pruning performs steadily well given enough time, but also in rare cases make seemingly arbitrary move.

4 Possible Improvements

Improvement for alpha-beta pruning, in my opinion, is quite limited. The high branching factor and exponential growth of time required together determine that the agent can at most search 3 plies in Java implementation, though with a better utility function, one can anticipate that its performance will be better.

For UCT, simulation and selection strategies could be further investigated to improve performance. One can make the move choosing policy for simulation based on a utility function that can be computed quickly, and in addition to simulation, one can add a utility function that is more accurate and perhaps slower to calculate the utility of the leaf state and count that to the selection strategy. One may also choose algorithms other than UCB1 to achieve a better balance between exploitation and exploration of children nodes. One can also reuse the search tree generated from last decision making to retain previous knowledge, but this is not necessarily an improvement, as shown in the comparison section.

For the utility function, instead of hand-crafting one which requires expert knowledge, one can obtain a utility function using machine learning methods, such as neural networks and reinforcement learning, which do not rely on the designer's available knowledge of the game and can be developed and improved systematically.

5 Comparison

In the last section, I will compare alpha-beta pruning with UCT, and investigate the impact of different features of UCT on its performance. For conciseness, I will only illustrate several major results.

1 The following is the result of running 50 games playing UCT (with utility) against UCT (without utility), each plays half of the time as the first player.

	# wins as White	# wins as Black
UCT (with utility)	19	16
UCT (without utility)	8	5
# draws	2	

As the result shows, adding utility of each state to UCB1 is a big boost of performance compared to the standard UCB1 algorithm.

2 In some implementations of Monte Carlo tree search, for example, in the famous Go player AlphaGo [4], the agent would reuse the search tree from last move rather than rebuilding it every time, and the idea is to learn based on previous knowledge. The following is the result of running 50 games playing UCT (with utility, no tree reuse) against UCT (with utility, tree reuse applied), each plays half of the time as the first player.

	# wins as White	# wins as Black
UCT (no tree reuse)	15	17
UCT (tree reuse applied)	7	7
# draws	4	

This result, however, is quite unexpected and shows that with my implementation of UCT, reusing the previously built search tree would actually decrease the performance. A possible reason could be that if the previously built search tree is biased, this bias is inherited and maybe amplified by searching deeper on the tree, before it eventually converges back to the minimax value (which is impractical).

3 The following is the result of running 50 games playing UCT (with utility, no tree reuse, time constraint 2s) against alpha-beta pruning (max-depth 3, no time constraint), each plays half of the time as the first player.

	# wins as White	# wins as Black
UCT	8	4
alpha-beta pruning	20	16
# draws	2	

As the result clearly shows, alpha-beta pruning without time restriction performs much better than UCT.

4 The following is the result of running 50 games playing UCT (with utility, no tree reuse, time constraint 2s) against alpha-beta pruning (max-depth 3, time constraint 2s), each plays half of the time as the first player.

	# wins as White	# wins as Black
UCT	15	2
alpha-beta pruning	23	10
# draws	0	

As can be seen, with the time constraint imposed on alpha-beta pruning, there is a notable drop of performance of it, though it is still sufficient to beat the UCT agent most of the time.

5 The following is the result of running 50 games playing the hybrid of UCT (with utility, no tree reuse, time constraint 2s) and alpha-beta pruning (max-depth 3, time constraint 2s) against alpha-beta pruning. The hybrid player uses alpha-beta pruning before the 7-th turn, uses UCT from 7-th to 14-th turns, and at last uses alpha-beta pruning again.

	# wins as White	# wins as Black
hybrid	4	2
alpha-beta pruning	20	21
# draws	3	

This result shows that this agent has an even lower winning rate against the alpha-beta player than the UCT agent. A hybrid agent is initially considered, since during early games alpha-beta pruning is usually fast, and UCT can make up for mid game. Due to this observation, this hybrid agent is not chosen as the final agent.

6 And last, the following is the result of running 50 games playing my agent (white UCT, black alpha-beta pruning) against the random player.

	# wins as White	# wins as Black
my agent	25	25
random agent	0	0
# draws	0	

The agent is obviously better than a random agent.

References

[1] Stuart J. Russell, Peter Norvig, Artificial Intelligence: A Modern Approach Third Edition. P165-167.

[2] L. Kocsis, C. Szepesvari, Bandit based Monte-Carlo Planning http://www.computer-go.info/resources/bandit.html

[3] Buescher Niklas, On Solving Pentago https://www.ke.tu-darmstadt.de/lehre/arbeiten/bachelor/2011/Buescher_Niklas.pdf

[4] Silver, D., Schrittwieser, J., Simonyan, K. et al. Mastering the game of Go without human knowledge. Nature 550, 354–359 (2017). https://doi.org/10.1038/nature24270