Reinforcement Learning: Monte Carlo Tree Search

Xiao (Ariel) Liang
s2614693

Chang Liu
s2560712

Renchi Zhang
s2590433

March 2020

1 Introduction

Monte Carlo Tree Search (MCTS) facilitates decision-making by stochastic sampling for strategic games where the heuristic planning proves infeasible or works unsatisfactorily. In this assignment, we implement an MCTS algorithm on the Hex game and test its functioning in a variety of ways. In order to compare the performance of MCTS against the more conventional, deterministic Minimax algorithm, Elo scores are adopted to delineate one-to-one play-outs between them two and generate visual assessment. We also design and conduct experiments to analyze the influence of tuning hyperparameters, namely, the number of simulations, N, and the exploration/exploitation parameter, C_p, upon the performance of MCTS.

2 Implementation

Underpinning our implementation of Monte Carlo Tree Search(MCTS) is the principles in Chapter 5 of *Learning to Play* [1], as well as an example of pseudo-code. We concretely constructed a tree data structure and its node class, and adjusted a few details on the MCTS operation in our own code. The implementation consists of three parts in three separate files — the data structure of MCTS tree, the search procedure and four MCTS operations, and numerous tests of MCTS. Details about them three will be introduced in the following sections. Moreover, we will discuss the problems we came across while implementing the algorithm and our thoughts about improving it.

2.1 Search Tree

The Monte Carlo search process is put into effect by a search tree in that each state of the game will be recorded as a node of the search tree, and its children nodes keep track of all potential status after a move. We implement such data structure with a Node class in the file *MCTS.py*. Every node object has several attributes and associated methods to update these attributes. Table 2.1 has included detailed descriptions of those features.

MCTS starts with a root node, which is initialized by the given state of the original Hex board. While searching deeper iteration by iteration, sub nodes are added to the parent node in the action of expending. The nodes' properties will also be updated in this process. What happens to the search tree will be elaborated in the next section.

2.2 Monte Carlo Tree Search

MCTS could be seen as a non-recursive process in that its four core operations, that is, selection, expansion, play-outs, and back-propagation, would be checked whether to skip or not in that specific order during the procedure. Within each iteration, the algorithm starts with selecting

attribute	description
parent	a node object; updated by add_child() in expanding
children	a list of node objects; updated by add_child() in expanding
state	a board object; updated by set_state() in expanding
last_move	a tuple object; updated while expanding by set_last_move() in expanding
move_color	an integer; updated while expanding by set_move_color() in expanding
visit_times	an integer; updated by add$_{v}ist_{t}imes()inbackpropagation$
quality_value	an integer; updated by add$_{q}uality_{v}alue()backpropagation$

Table 2.1: Attributes of the node object

the most promising candidate of next move from the current state. The key is how to evaluate the value of candidate move, and one of the convenient tools is Upper Confidence bounds applied to Trees(UCT) algorithm. UCT guides through the selection by tweaking the trade-off between exploration and exploitation. Because we want to secure the moves with demonstrated quality and are also interested in unknown, potential best moves, the UCT algorithm is adopted to integrate the win rate of a node with the number of times it has been visited. Less frequented node is compensated in such a manner that unknown potential is also scored other than proved success.

Selection is always conducted when all candidates of a new state after the next move are recorded in the search tree as node objects. The new state with selected move will be seen as the current state for the next selection. The corresponding information of new state is recorded

When there are new state not recorded in the search tree after a selected move, expend operation is applied to record the new state by creating a new sub node as a child of the parent node, and update its state, last move and last move color.

After expending a new node in the search tree, play-out action is conducted by take randomly moves from the current state of the new node until gameover. What we did here is to choose a random move from all possible moves on the board and exchange the move color after one random move.

When play-out is over, if the root player win, the reword is 1. Otherwise the reword is 0. Reword is used to update all the quality values of current node and parents of current node. Also, visit times of current node and its parents are added by 1.

After these 4 operations, we will see the sub node with highest visit times as the best sub-node, and choose the last move of this sub-node as the predicted next move.

2.3 MCTS Test

To test MCTS, we implemented 3 different methods. The first method is play against it manually in our interface. You can run the `interface.py`, enter a board size and choose mode 3 to run this experiment.

The second method we use is endgame. We set two endgames to see if the MCTS could give our expected result or not. These two test cases are shown in figure 2.1 and figure 2.2.

In endgame 1, position 1 and position 2 are equally valuable. So we want to see MCTS could always choose the next move from these two positions. Here we define that if the output of MCTS is not in these two positions, we would see the test failed. We set the simulation number to 500 and tested this case for 1000 times. The result shows that only 7 times the MCTS chose moves out side of the expected position. When we increased the simulation number to 1000, the MCTS didn't fail any test in our experiment.

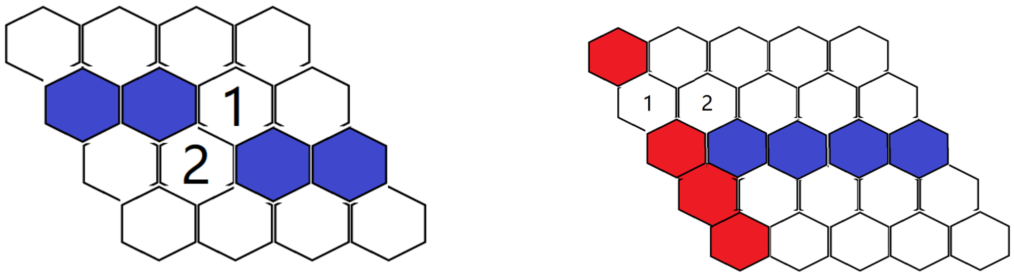


Figure 2.2: Endgame 2

In endgame 2, MCTS agent is the blue player. It's obvious that position 1 and position 2 are more valuable than other positions on the board, and position 1 here is more valuable than position 2. What we want to see is MCTS could always choose position 1 or position 2 as the next move, and choose position 1 more frequently than choose position 2. We run the experiment with 1000 simulations for 1000 test. The result we got is the MCTS chose position 1 919 times and chose position 2 81 times. The result shows that MCTS believes position 1 and position 2 are both valuable positions and position 1 is much better than position 2. The third method we use to test MCTS is run it against the minmax player.

To test MCTS on endgames and against minmax player, you may call the test functions in `test_MCTS.py` or just delete the comments in the main entry.

2.4 Discussions on MCTS

In our implementation, at first we tried to use win rate as the measurement to decide the final step after all iterations. In the iterations, we still use UCT value to select which node is going to be visited. Because we suppose that when the number of simulation is relatively large, after all iterations, the win rate of one node is convincing than visited times. But in our experiments, we find on our endgame 1, the final selection by win rate failed 18 times in 1000 tests while the final selection by visited times failed 7 times. Result shows that win rate is a worse measurement than visited times for the last selection. It also suggests that there exists some sub-nodes, whose visited times are low but win rate are high. These nodes are less likely to be the best move.

3 Experiments

3.1 Experiment Setting

In the experiment we let alpha-beta player to play against MCTS player, which is implemented in *experiment_mct_idtt.py* file 's *player1_vs_player2* function. Each time the game finished, we update two player's elo score computed by adopting Trueskill package[2]. From *experiment_mct_idtt.py* line 77-80, you can vary the specific experiment settings – board size, whether the alpha-beta utilizing the transposition table, the number of rollout of MCTS, and the number of total matches that two players will play to get the elo graph. Under each specific number of rollout of MCTS, we let it play against with alpha-beta player with depth3 and depth4.

We carry out the experiment on the board with size from 2*2 to 6*6.Meanwhile we set the number of games two players to battle as 20, as firstly we tried 12 as the Trueskill official website suggests, we find the final elo score has not been stable enough in many cases. For each board size we try to find the threshold of MCTS's number of rollout where MCTS player completely beat up alpha-beta overall. Meanwhile, we carry out the experiment limiting two players to spend the same amount of time playing, and check two players' performance.

Meanwhile, we carry out the experiment limiting two players to spend the same amount of time playing each move, and check their relative performance. We choose to limit the total running time for two players while not focusing on a single move, to smooth the influence of randomness in the game. To achieve this, we first let both player to play with random player to figure out the time 2 player needed respectively, as the time for random player to make decisions could almost be neglected. Then we match two players' time to the same level, and let them fight against each other. After analysing, we choose to set the time to 10s and 200s per game.

3.2 Run Experiments

To run the experiment you can run the *experiment_mct_idtt.py* python file directly. For each combination of the experiment setting, you can find the result in the newly created folder where the name of the folder is of the form "board6-20s-ab_with_idtt-mct_10000", which in order tells the board size, the number of total matches that two players played, player alpha-beta is equipped with transposion table or not, and finally the number of MCTS player's number of rollout. In each folder you can see 4 graphs with the name in the form of "player1-player2" (player1 means who is the first one to play). The elo graphs we get in the experiments could be found in the github [1].

3.3 Experiments results and discussion

board size	MCTS vs d3	d3 vs MCTS	MCTS vs d4	D4 vs MCTS
3*3	4	50	50	\
4*4	4	50	100	500
5*5	4	50	1000	800
6*6	3	50	10000	1000

The number in the table denotes the number of MCTS's rollout. After this number of rollout, the elo score of MCTS and alpha-beta (with depth3 or 4) is nearly stable, and MCTS's score is higher than alpha-beta's.

Table 3.1: Experiment table

From the table 3.1 we can see the number of rollout of MCTS needed to beat alpha-beta depth3 player is relatively stable on different size of the board. When depth3 player plays first, it needs MCTS to take more rollout to win.

When it comes to MCTS to play against alpha-beta depth4, things are a bit more difficult. For the board of size 3*3, even when we increase MCTS's rollout to 10,000, MCTS still could not beat alpha-beta depth4 player. We think it's due to on 3*3 board there is a critical position which is the center of the board. When alpha-beta player choose this position, it is almost unstoppable.

Then for board 4*4, the number of MCT's rollout needed to beat depth4 follow the rule from when MCTS play against depth3. However, when it comes to board size 5*5 and 6*6, a strange phenomenon shows up that – when depth4 plays first, the number of rollout MCTS needs is smaller than when itself plays first. We haven't figured out the possible reasons yet.

For board 6*6, we also limit the time for each player to choose their own best move. From 3.2 we can see when we fix the decision time (10s/200s) for both player, MCTS player always wins.

[1] `https://github.com/blackholebug/LIACS_RL/tree/master/A2/Elo%20graphs%20results`

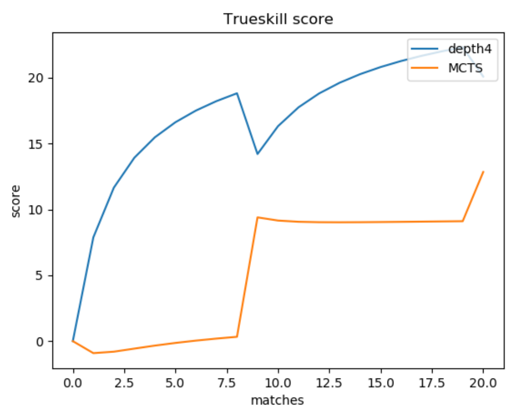
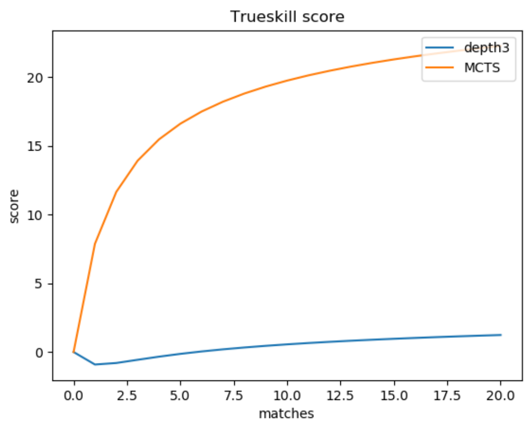
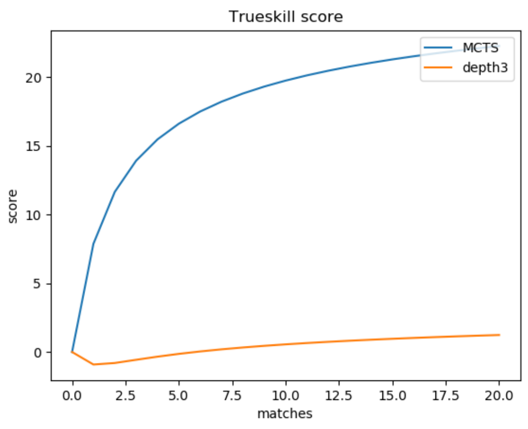


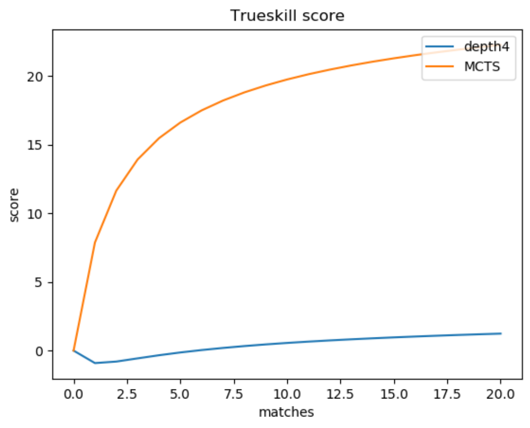
Figure 3.1: special case: depth4 vs MCTS(rollout 10000) on board 3*3



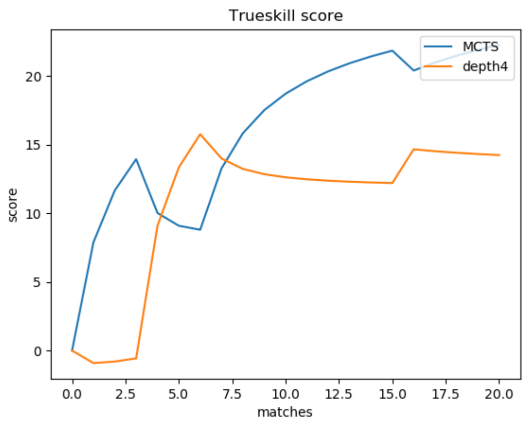
(a) Alpha-beta vs MCTS within 10 sec



(b) MCTS vs Alpha-beta within 10 sec



(c) Alpha-beta vs MCTS within 200 sec



(d) MCTS vs Alpha-beta within 200 sec

Figure 3.2: Elo graph of competition with fixed time on 6*6 board

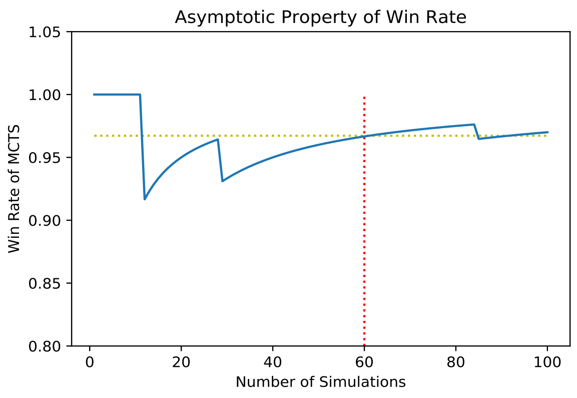


Figure 4.1: Winning Frequency

Board Size	N=500	N=1000	N=3000	N=5000
5	7.37	13.30	38.61	62.96
6	17.5	33.45	91.92	136.18
7	52.77	85.73	272.59	386.29
8	116.6	197.68	615.78	809.11
9	229.25	418.75	1585.73	1688.72

Table 4.1: Estimated seconds for one playout

4 Hyperparameters Tuning

4.1 Experiment Design

We decide to simulate play-outs between an MCTS algorithm and a random player for each pair of (N, C_p), evaluate the performance of MCTS, and find the optimal value of C_p for each N. To start with, performance would be measured by how likely it beats a random player on a medium-size board, as playing against an $\alpha - \beta$ algorithm should be computationally expensive and restrict the possibilities of (N, C_p) that we will be able to investigate.

The second decision is setting the board size to be 6, in which the influence of such pure luck as who plays first has diminished and the strategic nature of Hex starts to dominate. In a preliminary trial, the MCTS player wins all the time on a 5-by-5 board when $N = 3000$, suggesting that the board size is too small to be informative.

In order to determine the number of simulations for each pair of (N, C_p), with which the frequency of MCTS winning would be calculated, we simulate 100 play-outs on a 6-by-6 board, with $N = 500$ and $C_p = 1$, to approximate when the winning frequency converges. The choice of N and C_p is based on that (i) $C_p = 1$ is a baseline that must be considered, representing numerical balance between exploration and exploitation during the search; (ii) at least 500 iterations are necessary for a non-small board, and the winning frequency is expected to converge faster if N is larger, with more information collected by MCTS and the opponent still playing equally randomly. According to Fig 4.1, it is safe to extract the winning frequency from 60 simulations for each pair of (N, C_p).

The experiment design then comes down to what values of the two hyperparameters to investigate, and the decision is made in two steps. First, the research by Kuipers et al. suggested $C_p \in [0.5, 1.5]$ to be a reasonable range and 0.1 a practical increment[3]. Next, we settle down to $N = 500, 1000, 3000, 5000$ due to computational feasibility. For example, ten play-outs with $(N, C_p) = (5000, 1)$ are simulated, and each one takes 136.18 seconds on average (Table 4.1); multiplied by 60 simulations per C_p and 21 C_p values to traverse per N, the entire procedure of finding optimal C_p for $N = 5000$ would roughly require 48 hours.[1]This already hits the upper

	Cp	N=500	N=1000	N=3000	N=5000
0	0.50	1.000000	1.000000	1.000000	1.000000
1	0.55	1.000000	1.000000	1.000000	1.000000
2	0.60	0.933333	1.000000	1.000000	1.000000
3	0.65	0.916667	1.000000	1.000000	1.000000
4	0.70	0.950000	0.950000	1.000000	1.000000
5	0.75	0.966667	0.933333	1.000000	1.000000
6	0.80	0.966667	1.000000	0.983333	1.000000
7	0.85	0.950000	0.933333	1.000000	1.000000
8	0.90	0.916667	0.916667	1.000000	0.983333
9	0.95	0.950000	0.933333	0.983333	1.000000
10	1.00	0.933333	0.916667	1.000000	0.983333
11	1.05	0.883333	0.950000	1.000000	1.000000
12	1.10	0.900000	0.900000	1.000000	1.000000
13	1.15	0.950000	0.933333	0.966667	1.000000
14	1.20	0.883333	0.916667	0.950000	0.966667
15	1.25	0.916667	0.983333	0.966667	0.950000
16	1.30	0.950000	0.916667	0.950000	1.000000
17	1.35	0.983333	0.966667	0.900000	0.983333
18	1.40	0.900000	0.933333	0.933333	0.933333
19	1.45	0.933333	0.950000	0.966667	0.966667
20	1.50	0.933333	0.883333	0.916667	0.983333
Number of Consecutive Ones		2	4	6	8

Table 4.2: Winning Frequencies for (N, C_p) pairs: The larger the N, the more consecutive values of C_p at which the MCTS algorithm can beat the random player in 100% of simulated play-outs.

bound of pragmatical computational effort, and thus we do not consider any larger N.

Note that in practice, N is often not a parameter which we will be able to tune freely but rather a constraint of the game (e.g how soon a player is required to make the next move). Because the hyperparameters tuning task has been reframed as optimizing the MCTS performance by tweaking C_p for a given N, it is sensible to treat C_p candidates as "continuously" as possible but merely include four N values.

4.2 Results and Interpretation

Simulating 60 play-outs between an MCTS algorithm and a random player on a 6-by-6 Hex board, for each pair of (N, C_p) and computing the frequency that the former wins, we barely find any single optimal C_p for each N but a range of C_p values with which MCTS is almost surely winning. The range starts from the left of $[0.5, 1.5]$ at small N, implying the advantage of exploitation over exploration, and grows significantly wider to the right as N becomes larger; in other words, if the state space is to be searched sufficiently, even in a random manner, the amount of gathered information would make tuning C_p less critical. For example, when $N = 500$, only $C_p \in [0.5, 0.6]$ could yield the best performance, but every $C_p \in [0.5, 1.15]$ should suffice when $N = 5000$.

In addition, both Table 4.2 and Figure 4.2 display the overall reliability of the MCTS algorithm when N is large in that the winning frequency fluctuates less drastically in general.

[1]Because the computation for each pair of (N, C_p) is independent to each other, we split the task into simultaneous processes on multiple computers so as to shorten the computing. Results are integrated afterwards.

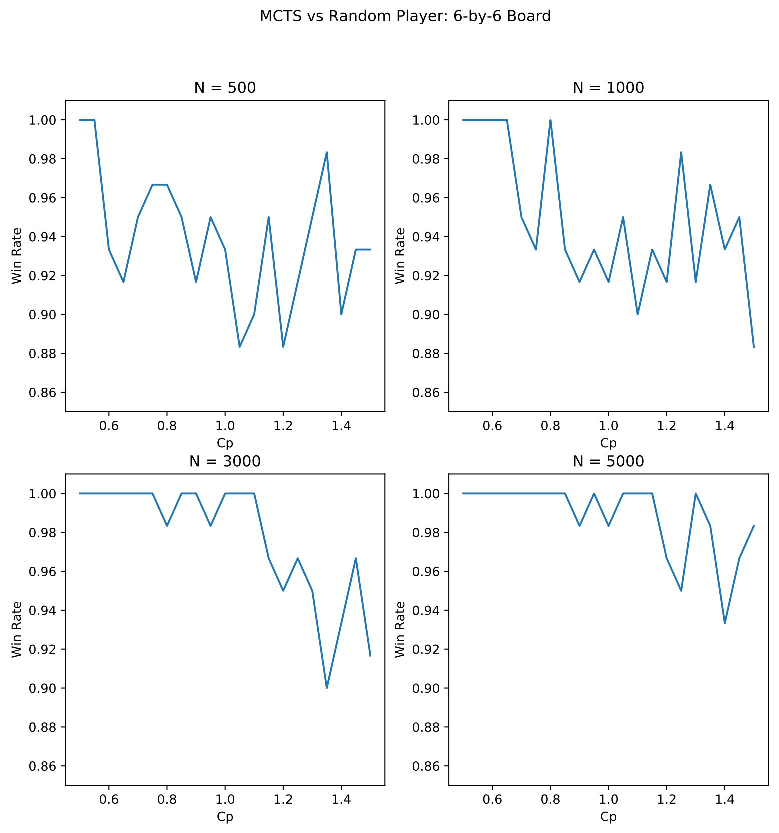


Figure 4.2: Although there is no statistical framework to infer how significantly the winning frequencies fluctuate, it makes intuitive sense that slight deviations from 100% on the left of each plot still reflect the same winning pattern.

5 Conclusion

This report documents the implementation, the performance evaluation, and the hyperparameter tuning of the Monte Carlo Tree Search algorithm on the Hex game. Not surprisingly, we find that MCTS fails to generate best moves when N, the number of searching iterations, is relatively small and information collected insufficiently. However, it gradually progresses towards the ideal performance as N grows larger. Our experiments also show that, regardless of board size, MCTS readily beats the $\alpha - \beta$ algorithm of search depth 3 within a few roll-outs; yet when the search depth increases to 4, who plays first has decisive influence. Interestingly, MCTS tends to win more often on bigger board when playing second. And finally, by evaluating the performance of MCTS for pairs of (N, C_p), we conclude that the tradeoff between exploration and exploitation in the searching strategy must be tweaked in specific to N. In particular, exploitation is markedly preferred when N is small, and the greater the N, the less sensitive the performance to the value of C_p. If we were to explore this topic further, adding a transition table and trying one of those revised UCT formulas would both be great starting points.

References

[1] Aske Plaat. Learning to play, reinforcement learning and games. 2020.

[2] Microsoft. TrueSkill documents. `https://trueskill.org/`. [Online;].

[3] Jan Kuiper Jos AM Vermaseren, Aske Plaat and Jaap van den Herik. Carlo tree search for finding better multivariate horner schemes. page 3, 02 2013.