On the Road to Perfection? Evaluating Leela Chess Zero Against Endgame Tablebases

Rejwana Haque[1], Ting Han Wei[1], and Martin Müller[1][0000−0002−5639−5318]

University of Alberta, Edmonton, Canada
{rejwana1,tinghan,mmueller}@ualberta.ca

Abstract. Board game research has pursued two distinct but linked objectives: solving games, and strong play using heuristics. In our case study in the game of chess, we analyze how current AlphaZero type architectures learn and play late chess endgames, for which perfect play tablebases are available. We study the open source program Leela Chess Zero in three and four piece chess endgames. We quantify the program's move decision errors for both an intermediate and a strong version, and for both the raw policy network and the full MCTS-based player. We discuss a number of interesting types of errors by using examples, explain how they come about, and present evidence-based conjectures on the types of positions that still cause problems for these impressive engines.

Keywords: AlphaZero learning · Computer chess · Leela Chess Zero.

1 Introduction

The AlphaZero algorithm [12] has demonstrated superhuman playing strength in a wide range of games, such as chess, shogi, and Go. Yet as powerful as neural networks (NNs) are at move selection and state evaluation, they are not perfect. Judging from the varied outcome of self-play games in deterministic complete information games such as chess and Go, even the best AlphaZero-style players must still make mistakes. We investigate this gap between strong play and perfect play. To analyze how these modern programs learn to play sophisticated games, and also to test the limits to how well they learn to play, we turn to a sample problem that has known exact solutions. While the full game of chess has not yet been solved, exact solutions for endgames up to seven pieces have been computed and compiled into endgame tablebases. We use the AlphaZero-style open-source program *Leela Chess Zero* (Lc0) to analyze chess endgames. We develop a methodology and perform large-scale experiments to study and answer the following research questions:

- How do stronger and weaker networks differ for predicting perfect play?
- How does the search in Lc0 improve the prediction accuracy for endgames?
- How do stronger policies improve search results?
- How does reducing the search budget affect the correctness of Lc0?
- Which is easier to predict, wins or draws?
- How well does Lc0 recognize wins and losses?

– Which kinds of endgame positions are easy and hard to learn? How does that change with more learning?
– Are there cases where search negatively impacts move prediction? If such cases exist, why do they occur?

2 Background

In previous work [11,6], perfect play has been compared against heuristic engines.

2.1 Chess Endgame Tablebases

Chess is a game in which the state space and game tree complexity is reduced as the game progresses and pieces are captured. Chess endgames are sub-problems in which the full rules of chess apply, but only a reduced set of game pieces remains on the board. While the game of chess itself has not been solved to date, endgames of up to seven pieces have been solved and are publicly available [9]. A database of such endgame solutions is referred to as an endgame tablebase. A solution for each position includes the outcome of the game given perfect play for both players, the optimal moves that each player must make to reach that outcome, and specific metrics such as the number of plies (moves by one player) required to reach the outcome. There are advantages and disadvantages to each kind of metric. In this paper, we use the *depth to mate* (DTM) metric, which is the number of plies for a win or loss, assuming the winning side plays the shortest way to win, and the losing side the longest to lose [4].

Endgame tablebases hosted online [5,10] differ in storage size and metrics. Tablebase generators are often also available. Among these, the Syzygy [10] and Gaviota [1] tablebases are free and widely used.

2.2 Leela Chess Zero

Leela Chess Zero (Lc0) is an adaptation of the Go program Leela Zero for chess. Both programs are distributed efforts which reproduce AlphaZero for chess and Go, respectively. Volunteers donate computing resources to generate self-play games and optimize the NNs. Relying solely on community resources and efforts, in 2020 Lc0 surpassed AlphaZero's published playing strength in chess [7].

Like AlphaZero, Lc0 also takes a sequence of consecutive raw board positions as input. It uses the same two-headed (policy and value) network architecture as AlphaZero. Similar to AlphaZero, the policy head output guides the Monte Carlo tree search (MCTS) as a *prior probability*, while the value head output replaces rollouts for *position evaluation*. Over time the developers of Lc0 introduced enhancements that were not in the original AlphaZero. For example, an auxiliary output called the *moves left head* was added to predict the number of plies remaining in the current game [3]. Another auxiliary output called the *WDL* head separately predicts the probabilities that the outcome of the game is a win, draw, or loss [8]. Lc0 uses two distinct training methods to generate different types of networks that differ in playing strength. T networks are trained by self-play, as in AlphaZero, while J networks are trained using self-play game records generated from T networks. J networks are stronger and are used in tournament play.

3 Analyzing Chess Endgames with Leela Zero Chess

3.1 Tablebase Dataset Pre-processing

In order to evaluate the performance of a chess program in terms of finding exact solutions, a ground truth to compare against is needed. For that, we use chess endgame tablebases which describe perfect play of all positions up to 7 pieces on the board. In this section we describe how to pre-process the tablebases.

We choose the open-source Gaviota tablebases as the source of ground truth. We compare the perfect play against the tested program's choice for all unique legal positions in all nontrivial three and four piece endgames. Also, we took the winning and drawing positions where there are more than one possible outcome.

Since the Gaviota tablebase is indexed and compressed, the following steps are performed to create an iterable list of positions:

1. We use the method of Kryukov [5] to enumerate all unique legal positions and store positions in the FEN format. Uniqueness refers to treating symmetric positions as one; legality is checked using normal chess rules.
2. We use tools from python-chess [2] to extract the following information from the Gaviota tablebase for each enumerated position: The position in FEN format, lists of all winning, drawing and losing moves, the win-draw-loss status, the DTM score, and the decision depth (see below).
3. The data for each endgame type is stored in MySQL for easy access.

We use the term *decision depth* to categorize positions in an endgame tablebase. For a winning position, the decision depth is simply the DTM score. For drawing positions where there are also losing moves, the decision depth is the highest DTM after a losing move.

3.2 Choice of AlphaZero Program and Its Parameters

We chose Lc0 0.27 as our AlphaZero-style program in our analysis, because it is both publicly available and strong. Lc0 has evolved from the original AlphaZero in many ways, but with the proper configuration, it can still perform similarly. We use two specific settings and leave everything else in the default configuration.
Disabling Smart Pruning: AlphaZero selects the node with the highest PUCB value during MCTS [13]. However, Lc0 does not always follow this behaviour. This is due to smart pruning, which uses the simulation budget differently: it stops considering less promising moves earlier, resulting in less exploration. We set the Lc0 parameter `-smart-pruning-factor=0` to disable smart pruning.
Single-threaded Search: For consistent analyses, we prefer the engine to be deterministic between different experiment runs. Multi-threading introduces random behaviour, so we run the engine with a single thread for consistency.

Lc0 supports a variety of NN backends. Since we used Nvidia Titan RTX GPUs for our experiments, we chose the `cudnn` backend.

Many network instances are publicly available. For this research we chose two specific snapshots of the T60 network generation:
Strong network: ID 608927 with (self-play) ELO rating 3062.00, which was the best performing snapshot up to May 2, 2021
Weak network ID 600060, rating 1717.00, were the initial weights of this generation after 60 updates starting from random play.

Table 1: Total number of mistakes by the policy net and MCTS with 400 simulations, using strong and weak networks.

EGTB	Total Positions Tested	Weak Network		Strong Network	
		Policy	MCTS-400	Policy	MCTS-400
KPk	8596	390	13	5	**0**
KQk	20743	109	**0**	**0**	**0**
KRk	24692	69	**0**	**0**	**0**
KQkq	2055004	175623	12740	3075	36
KQkr	1579833	141104	3750	4011	46
KRkr	2429734	177263	6097	252	**0**
KPkp	4733080	474896	41763	20884	423
KPkq	4320585	449807	46981	6132	13
KPkr	5514997	643187	60605	13227	196

4 Experiments

4.1 Move Prediction Accuracy for Basic Settings

We evaluate the move decisions of Lc0 for all non-trivial three and four piece endgame positions. We define a *mistake* as a move decision that changes the game-theoretic outcome, either from a draw to a loss, or from a win to not a win. Any outcome-preserving decision is considered correct. So the engine does not need to choose the quickest winning move. Accuracy is measured in terms of the number or frequency of mistakes, over a whole database.

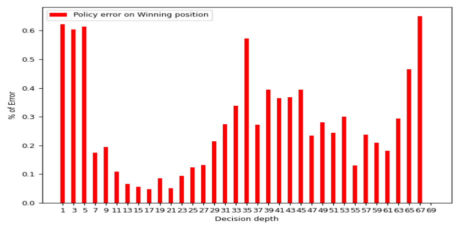
Table 1 shows the number of mistakes, for each three and four piece tablebase, for both the weak and the strong network. Results are shown for both the raw network policy, and for full Lc0 with 400 MCTS simulations. From the table it is clear that both network training and search work very well. The strong net makes significantly fewer mistakes, and search strongly improves performance in each case where the network alone is insufficient. For all three piece positions, 400 MCTS simulations are enough to achieve perfect play. In four piece endgames, a small number of mistakes still remain under our test conditions.

Effect of Search Budget on Winning and Drawing Positions To analyze how deeper search influences accuracy, we compare search budgets of 0 (raw policy), 400, 800, and 1600 simulations per move decision. We call these settings MCTS-0 = policy, MCTS-400, ... We chose these relatively settings considering our limited computational resources. Deeper search consistently helps for all of these tablebases.

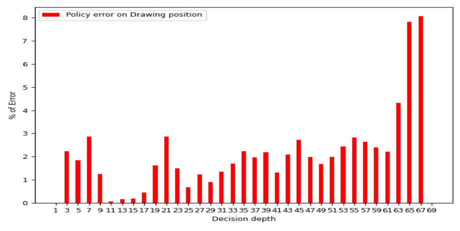
The error rate, defined as the fraction of mistakes in a set of positions, is shown separately for the sets of winning and drawing positions in each tablebase in Table 2. In many of these datasets, decisions are more accurate for draws than for wins. The main exception is KQkr, which contains only a small fraction (3.4%) of draws. The error rate for those draws is very high at 2%.

Table 2: Error rate for winning and drawing move predictions for MCTS with search budgets of 0 (raw policy), 400, 800 and 1600 simulations.

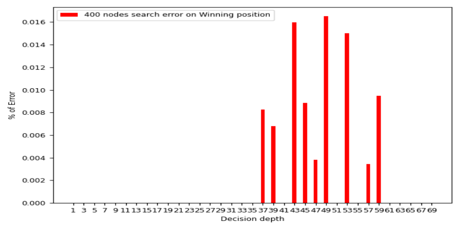
EGTB	% of Error							
	Winning Positions				Drawing Positions			
	0	400	800	1600	0	400	800	1600
KPk	8.00e−2	0.00	0.00	0.00	0.00	0.00	0.00	0.00
KQk	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
KRk	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
KQkq	2.96e−1	3.21e−3	1.61e−3	0.00	2.74e−2	5.35e−4	8.03e−4	5.35e−4
KQkr	1.53e−1	1.41e−3	4.01e−4	0.00	2.01	2.93e−2	1.41e−2	1.17e−3
KRkr	2.33e−2	0.00	0.00	0.00	4.19e−3	0.00	0.00	0.00
KPkp	4.39e−1	1.02e−2	3.30e−3	1.46e−3	4.45e−1	6.18e−3	1.51e−3	3.29e−4
KPkq	1.18e−3	2.94e−6	1.34e−6	10.07e−6	2.94e−3	3.42e−6	0.00	0.00
KPkr	2.02e−1	4.11e−1	3.25e−3	4.92e−3	1.22e−3	1.81e−3	2.88e−4	1.31e−3



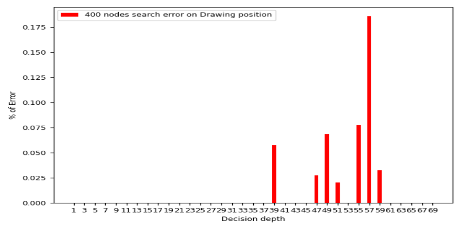
(a) Policy errors in winning positions.



(b) Policy errors in drawing positions.



(c) MCTS-400 errors in winning positions.



(d) MCTS-400 errors in drawing positions.

Fig. 1: Error rate for each decision depth in the KQkr tablebase.

4.2 Performance at Different Decision Depths

In this experiment, shown in Figure 1, we measure the error rate separately at each decision depth. We evaluate both the raw policy and MCTS-400 using the strong network. The figure shows that in contrast to raw policy, MCTS-400 only makes mistakes at higher decision depths. Policy mistakes at all shallow decision depths are completely corrected by search. At higher depths, some errors remain, but there is no simple relation between decision depth and error rate there.

Figure 2 shows that there is a relationship between the sample size at each decision depth and the error rate of the raw net. Each point in the figure corresponds to all positions of a specific decision depth in KQkr. The results in other four piece tablebases are similar in that fewer positions at a given depth correspond to more errors. They are omitted here for brevity. Figure 1(c)-(d) shows the corresponding results for MCTS-400. The engine only makes mistakes in positions with higher decision depths. Search can routinely solve positions with shallower decision depths regardless of the policy accuracy.

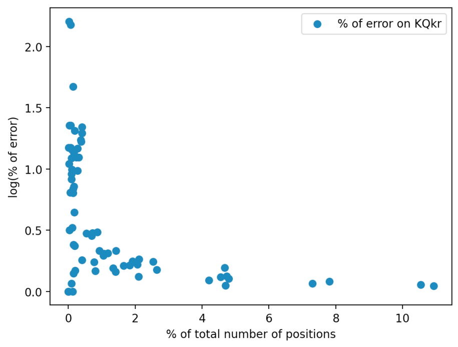


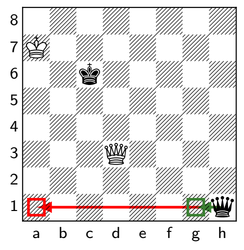
Fig. 2: Sample size (shown as percentage of the total number of positions) at each decision depth vs. raw policy error rate (in log scale) for the KQkr tablebase.

4.3 Case Studies: Interesting Engine Mistakes

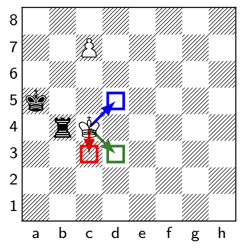
In this section we study a number of interesting cases where Lc0 makes mistakes. While analyzing these mistakes, there is a large amount of common expected behavior: search typically corrects policy inaccuracies, and larger searches correct errors that are still made by smaller searches. However, there are cases where search fails while the policy is correct. Several examples discussed below are shown in Figure 3. The correct moves are indicated in green and blue, while chosen incorrect moves are shown in red.

Policy Wrong, Search Correct: In Figure 3(a), **Qg1** wins but **Qa1** only draws. The network's prior probability (policy head) of the incorrect move **Qa1** (0.1065) is higher than for the winning move **Qg1** (0.0974). However, the value head has a better evaluation for the position after the winning move (0.3477) than the drawing move (0.0067). Therefore **Qg1** becomes the best-evaluated move after only a few simulations. Figure 4(a1-a4) shows details - the changes of Q, U, $Q+U$ and N during MCTS as a function of the number of simulations. At each simulation, the move with the highest UCB value ($Q+U$) is selected for evaluation. The Q value of a node is the average of its descendants' values. The exploration term U depends on the node's visit count N and the node's prior probability. For this example, while the exploration term $U(\mathbf{Qa1}) > U(\mathbf{Qg1})$ throughout, the UCB value remains in favour of the winning move. An accurate value head can overcome an inaccurate policy in the search.

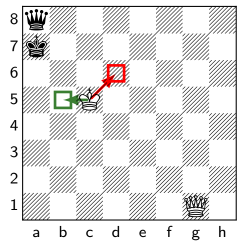
Policy and Search Both Wrong: In Figure 3(b), both **Kd3** and **Kd5** win, but both the raw network and the search choose **Kc3** which draws. **Kd5** has by far the lowest policy (0.2776) and value (0.3855), and its Q and N are consistently low, keeping it in distant third place throughout. Both the initial policy and value



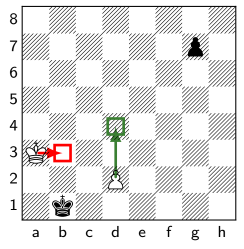
(a) Policy wrong, search correct



(b) Policy wrong, search also wrong



(c) Policy correct, smaller search wrong



(d) Policy correct, all search wrong

Fig. 3: Examples for different types of engine mistakes.

are higher for **Kc3** (0.3229 and 0.9144) than for the correct **Kd3** (0.2838 and 0.8501). We extended the search beyond the usual 1600 simulations to see longer-term behavior. The Q value of **Kc3** remains highest for 6000 simulations, while **Kd3** catches up, as shown in Figure 4(b1). MCTS samples all three moves with similar UCB values, but focuses most on the incorrect **Kc3**. At 1600 simulations, the inaccurate value estimates play a large role in incorrectly choosing **Kc3**. Beyond 6000 simulations, the Q value of **Kd3** keeps improving, and MCTS finally chooses a correct move at about 12000 simulations.

Policy Correct, Smaller Search Wrong: In Figure 3(c), **Kb5** wins while **Kd6** draws. The prior probability of **Kb5** is 0.0728, which is slightly higher than **Kd6**'s at 0.0702, but the value, at 0.2707, is slightly lower than **Kd6**'s at 0.2834. Figure 4(c1) shows that the Q value of **Kd6** is higher early on due to the value head. As search proceeds, this reverses since the values in the subtree of the winning move are better. In this example, MCTS overcomes an inaccurate root value since the evaluations of its followup positions are more accurate.

Policy Correct, Search up to 1600 Simulations Wrong: In the example shown in Figure 3(d), **d4** wins. Up to 1600 simulations, MCTS chooses the drawing move **Kb3**. The value of **Kb3** (0.1457) is higher than that of **d4** (0.1053), but the prior probability of **d4** (0.3563) is higher than **Kb3** (0.348). Figure 4(d1-d4) shows the search progress. The Q value of **d4** remains lower for longer than in the previous example in Figure 3(c). At around 1500 simulations, the UCB value of the correct move becomes consistently higher. This prompts the search to sample the correct move more, At 2200 simulations, the Q value of the correct **d4** spikes dramatically. At this point the search tree is deep enough to confirm the win, and from 2700 simulations on, the engine plays **d4**.

5 Summary and Future Work

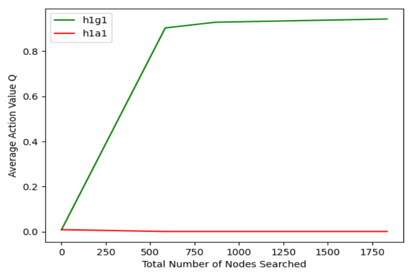
The important findings for three and four piece tablebases are: 1) NNs approach perfect play as more training is performed. 2) Search helps improve prediction accuracy. 3) The number of NN errors decreases for decision depths that have a higher number of samples. 4) Search increases the rate of perfect play with shallower decision depths. 5) Search corrects policy inaccuracies in cases where the value head accuracy is high. 6) Small-scale search may negatively impact accuracy in cases where the value head error is high. However, deep search eventually overcome this problem in the endgames we analyzed.

Future extensions of this study include: 1) Extend the study for larger endgame tablebases (with more pieces) to generalize our findings. 2) Perform frequency analyses of self-play training data to get the number of samples at each decision depth. 3) Analyze symmetric endgame positions to verify decision consistency. 4) Examine value head prediction accuracy and compare with policy accuracy. 5) Study the case where the program preserves the win, but increases the distance to mate. How often would the program run into the 50 move rule?

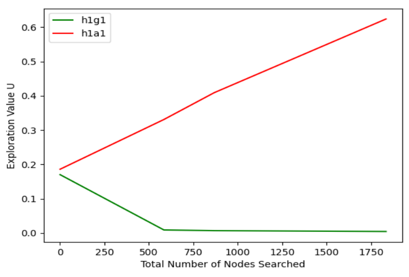
Acknowledgements: The authors gratefully acknowledge support from NSERC, the Natural Sciences and Engineering Research Council of Canada, and from Müllers DeepMind Chair in Artificial Intelligence and Canada CIFAR AI Chair.

References

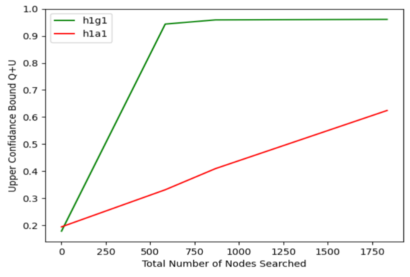
1. Ballicora, M.: Gaviota. `sites.google.com/site/gaviotachessengine/Home/endgame-tablebases-1`, accessed: 2021-06-06
2. Fiekas, N.: python-chess. `python-chess.readthedocs.io` (August 2014)
3. Forstén, H.: Purpose of the moves left head. `github.com/LeelaChessZero/lc0/pull/961#issuecomment-587112109`, accessed: 2021-08-24
4. Huntington, G., Haworth, G.: Depth to mate and the 50-move rule. ICGA Journal **38**(2), 93–98 (2015)
5. Kryukov, K.: Number of unique legal positions in chess endgames. `http://kirill-kryukov.com/chess/nulp/` (2014), accessed: 2021-08-30
6. Lassabe, N., Sanchez, S., Luga, H., Duthen, Y.: Genetically programmed strategies for chess endgame. In: Genetic and evolutionary computation. pp. 831–838 (2006)
7. Lc0 Authors: What is Lc0? `lczero.org/dev/wiki/what-is-lc0/`, accessed: 2021-08-24
8. Lc0 Authors: Win-draw-loss evaluation. `lczero.org/blog/2020/04/wdl-head/`, accessed: 2021-08-24
9. de Man, R.: Syzygy. `github.com/syzygy1/tb`, accessed: 2021-06-06
10. de Man, R., Guo, B.: Syzygy endgame tablebases. `syzygy-tables.info/`, accessed: 2021-06-06
11. Romein, J.W., Bal, H.E.: Awari is solved. ICGA Journal **25**(3), 162–165 (2002)
12. Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., et al.: A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. Science **362**(6419), 1140–1144 (2018)
13. Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., et al.: Mastering the game of Go without human knowledge. nature **550**(7676), 354–359 (2017)



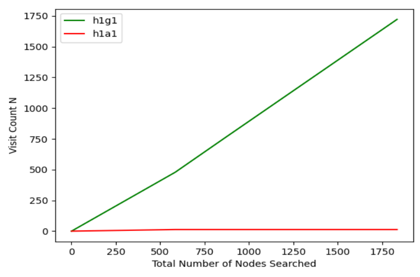
(a1) Average action value Q



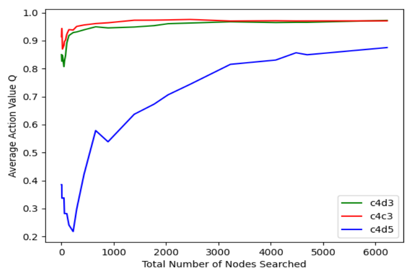
(a2) Exploration term U



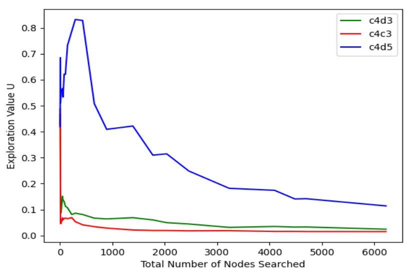
(a3) Upper confidence bound $Q + U$



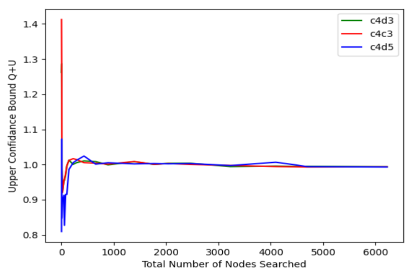
(a4) Visit count N



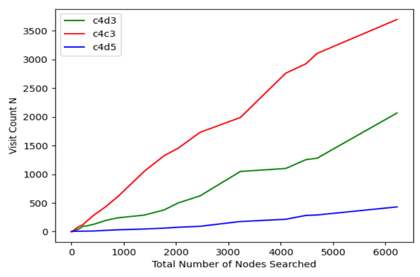
(b1) Average action value Q



(b2) Exploration term U

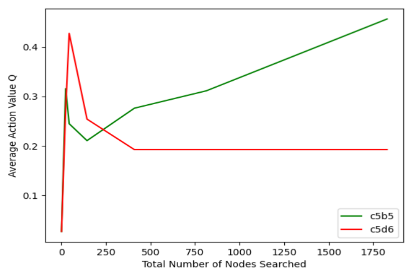


(b3) Upper confidence bound $Q + U$

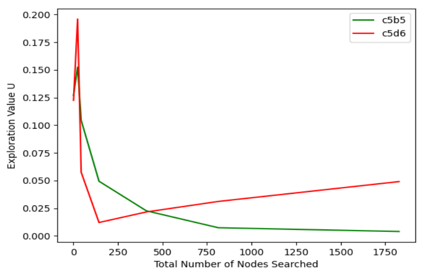


(b4) Visit count N

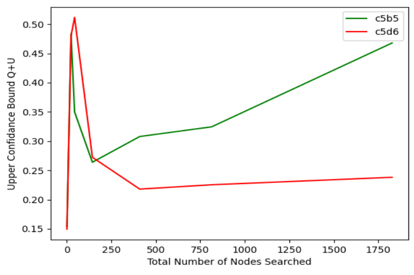
Fig.4: Page 1 of 2. Development of relevant terms Q, U, $Q + U$, N in UCB for Figure 3((a)-(b)).



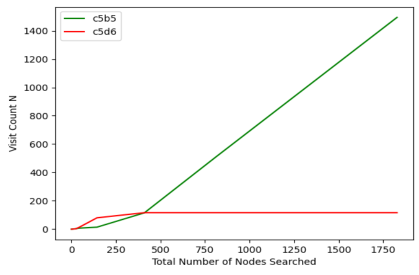
(c1) Average action value Q



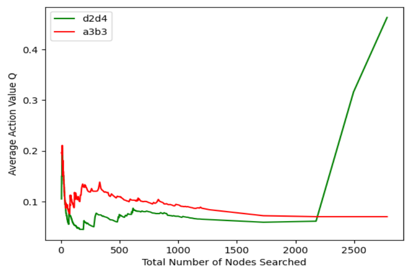
(c2) Exploration term U



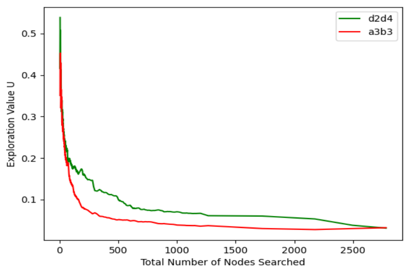
(c3) Upper confidence bound $Q + U$



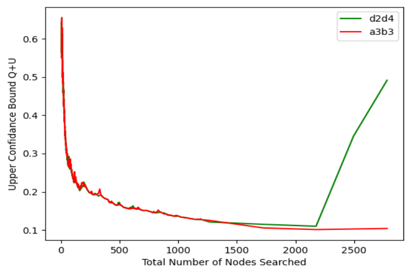
(c4) Visit count N



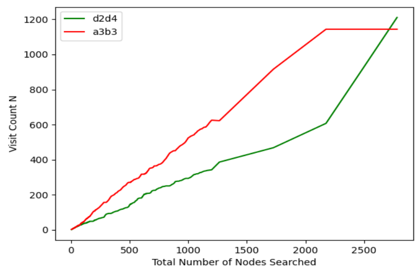
(d1) Average action value Q



(d2) Exploration term U



(d3) Upper confidence bound $Q + U$



(d4) Visit count N

Fig. 4: Page 2 of 2. Development of relevant terms Q, U, $Q + U$, N in UCB for Figure 3((c)-(d)).