Report on the Reproduction of Tree of Thoughts and Modifications

Chihsheng Jin
University of Rochester
`zjin22@ur.rochester.edu`

Abstract

(Code is available in this repo: https://github.com/ChihshengJ/tree-of-thought-llm.git)

In this report, I evaluated the performance of the Tree-of-Thoughts algorithm across various language models and multiple ablation studies. I found that in terms of 24 puzzles, the algorithm itself underperformed the standard prompt and sample strategy, as well as CoT prompting, by a large margin with GPT-3.5-turbo and the author's original settings. However, the performance of ToT itself in the paper matches the performance we observed. To improve the success rate, performance consistency, and inference speed, we tested three additional features: system prompts, truncating proposals, and program-aided evaluation for branch pruning. We found that incorporating the latter two can drastically speed up the inference and improve performance.

1 Introduction

In the original paper (Yao et al., 2024), the authors proposed a promising algorithm to improve the reasoning abilities of language models when given tasks with complex dependencies, resembling the strategy used by OpenAI's O1 model. Given that interpreting parent-child relationships would be much easier with intermediate results in 24 puzzles, I chose this as the task for my experiments. While analyzing the implementation of the algorithm, I noticed two aspects that could render this algorithm incapable of fulfilling the authors' promises.

First, the implementation does not maintain a tree structure for the intermediate results. After evaluating each candidate, it only takes the top k candidates to compute the next level of the tree, where k is a hyperparameter. In this sense, the structure of this implementation resembles a feed-forward neural network: the 'get_values' function acts as the activation function, and each request for output functions as a linear layer of the network. A severe drawback of this implementation is that when all candidates at a certain level yield values close to zero, it has no mechanism to trace back to their parent nodes and explore different routes. As a result, it often gives up.

The second observation is that the implementation is extremely slow. We observed that it could take up to 10 minutes to generate a failed attempt at solving a case with GPT-3.5-turbo, and it was even slower and more costly with GPT-4o. The main culprits are the proposal process and the lack of system prompts. The absence of system prompts made this implementation extremely vulnerable to the latest models, such as LLaMA-3.1-70B and GPT-4o-mini, which tend to return verbose generations and do not adhere strictly to the instructions, making the experiment nearly impossible to complete. Furthermore, the original implementation had no way to control the number of generations at each depth of the tree, sometimes allowing the number of proposals at one depth to exceed 50, which drastically increases the number of requests during the evaluation process.

2 Reproduction with GPT-3.5-turbo

To reproduce the results of game24 with different settings (standard-sampling, CoT, and ToT) in the original paper, I tried various language models via different APIs. I found that gpt-4 and gpt-4o were way too expensive (approximately 74$ per run) and slow, and gpt-4o-mini failed all the cases in one run with system_prompt (otherwise the output would be very verbose and hard to evaluate). Thus, for the sake of keeping the budget at a reasonable range, I chose GPT-3.5-turbo to reproduce the results. In the appendix of the paper the authors also shown the results from GPT-4 and GPT-3.5-turbo, so it would make a good comparison.

The results are close to that from the original paper, but what they did not report was the average accuracy of OI, CoT prompting with multi-

Methods	Average Any Success (%)	Average Success (%)
IO (best of 100)	58	8
CoT-SC (k=100)	50	4
ToT (b=5)	26	7

Table 1: We used the same parameters as the ones in the appendix to test whether GPT-3.5-turbo's performance matches the results in the paper. Average Any Success means in a pool of samples generated with the same input, if there's one success, we count this example as a success case, while the Average Success means average success rate per attempt.

ple candidates, which's been referred to as "Any Success". The any success rates of the two traditional methods were surprisingly high and their inference speed and cost were far superior than the ToT approach. In fact, we noticed that since the ToT approach introduced too much procedures in the prompting, it could be very vulnerable to wrong output in intermediate steps. During experiments, GPT-3.5-turbo constantly made mistakes in continuation, which include using wrong numbers, generate wrong answer for a correct tree of outputs, and make arithmetic mistakes.

3 Experiment on different models

Due to the fact that the numbers of token usage were extremely huge (we expected the responses to be all numbers and operators in formats we provided in the prompts) and APIs offered by different companies have strict usage limits, and I don't have any coverage over the expense on the projects, the off-the-shelf method in this paper did not work. In fact, we found that for most the models, the time usage for one case would be 323s in average, which defeats its own purpose in some sense. We found that when running the original code, there's possibility that one example would exhaust my daily token limit for one API. So we used system prompts and the program aided evaluator (we'll introduce them in the section 4) to test the performance on different large language models.

The results from three different models we used and the original result are shown in Table 2. We tested GPT3.5 turbo, Claude Sonnet 3.5, and LLaMa3.1 70B. We used a sample size of 7 for each pruning process (b in this paper), and all the other setting remained the same. The results shown that the performance of ToT prompting is highly dependent on the model. Smaller models that are less competent at reasoning would perform significantly worse than larger models. This aligns with our empirical findings because models that are not good at reasoning tend to perform badly on following the instructions(cite!) and arithmetics(cite!). It's also worth noting that during the experiment on LLaMa3.1-70B, we encountered at least 5 internal server errors. Due to the logging mechanism of the script, sometimes an almost succeeded tree would be forced to run again, resulting to a failure. Thus we expect the true success rate for LLaMa3.1-70B to be higher.

4 Additional Features

In this project I implemented 3 additional features to the algorithm to make it executable on a lower budget and realistic time constraint. They are system prompts, truncating length & early stop, and program-aided evaluator.

4.1 System prompts

Surprisingly, the experiments in the original paper did not use any system prompts for the tasks. The reason might be that it saves input tokens, but it certainly made the output excruciatingly unstable. It might worked on GPT4, but most of the other language models would generate many irrelevant tokens that hinder the post processing. For the game24 task, I wrote a system prompt for the generating stage of all prompting methods and one for the evaluation of the ToT. The prompts are in the Appendix. Table 3 shows how the system prompt affects the result of all prompting methods on GPT-3.5-turbo.

Table 3 shows that system prompts gave standard prompting and CoT prompting a significant increase, which suggests that the authors did not try their best to achieve the best performance for their baselines. However, the performance decreased to nearly zero for ToT. Upon further inspection over the log files, it seems like the system prompt made GPT3.5 generate more "*possible next step:*", which was part of the instructions of the original prompt. So the model might misunderstood the instruction

Model	**Average Any Success (%)**	**Average Success (%)**
GPT-3.5-turbo (symbolic evaluator)	26	7
Claude-Sonnet-3.5 (symbolic evaluator)	83	37
LLaMa-3.1-70B (symbolic evaluator)	59	19
GPT-4 (b = 5, from the paper)	74	?

Table 2: With the same b value, we evaluated the success rate and the average inference time of the three models. We used GPT-4 to generate the program-aided evaluator, the prompt is shown in the Appendix.

Methods	**system prompt**	**Average Any Success (%)**	**Average Success (%)**
IO (best of 100)	False	58	8
CoT-SC (k=100)	False	50	4
ToT (b=5)	False	26	7
IO (best of 100)	True	66	8
CoT-SC (k=100)	True	56	3
ToT (b=5)	True	2	6

Table 3: System-prompted vs. non-system-prompted performance across various methods

and adopted the wrong format, which made many output impossible to evaluate and branch further. The result shows that ToT does not necessarily perform better than other prompting methods and further prompt engineering for ToT to function effectively is needed.

4.2 Truncating length & Early stop

Truncating length and early stop are two functionalities I added to the program. I noticed that the original algorithm did not restrict the length of the proposals with regard to one previous step. For example, if you have 5 previous steps as candidates from the last depth of the tree, the model could generate 10 candidates for the first previous step, 4 for the second one, simply because it's hard to set a stop parameter. In the BFS loop I implemented a parameter *truncating length* that could control how many choices are actually used in the following evaluation, which could make evaluation much faster since in the original implementation of evaluation, the value of the evaluation would be a result of sampling from multiple language model outputs.

For early stop, I implemented a method in the task class that could evaluate whether the outcome could indicate an early stop of the search. In the original code, the whole BFS would not stop until all the steps were evaluated, so sometimes even when a correct answer has been generated, it would continue to explore other paths, which would be meaningless regarding the task.

The algorithm with truncating length and early stop being integrated is shown in Algorithm 1.

Algorithm 1 Description of the updated algorithm

```
Require: Tuple of four integer tuples $T = (t_1, t_2, t_3, t_4)$
Require: Number of iterations $step$
Require: truncating length $k$
Require: width of each depth $b$
Ensure: Final list of selected values $ys$
 1: Initialize $ys \leftarrow \emptyset$
 2: for $i \leftarrow 1$ to $step$ do
 3:     Set $early\_stop \leftarrow False$
 4:     for all $y$ in $ys$ do
 5:         if $is\_early\_stop(y)$ then
 6:             Set $early\_stop \leftarrow True$
 7:         end if
 8:     end for
 9:     if $early\_stop$ then
10:         Break loop
11:     end if
12:     $P_1 = ProposalWrap(T_p, ys)$
13:     $new\_ys = SampleFrom(LLM(P_1))$
14:     for all $new\_y$ in $new\_ys$ do
15:         $P_2 = EvaluateWrap(T_e, new\_y)$
16:         $v_new_y = SampleFrom(LLM(P_2))$
17:     end for
18:     Sort $new\_ys$ by $v_{new\_y}$ in descending order
19:     Select top $b$ $new\_ys$ to form $selected\_ys$
20:     Update $ys \leftarrow selected\_ys$
21: end for
22: return $ys$
```

The results from the experiment shows that setting truncating length at 8 without system prompt yielded a 0% success rate. It could be result from the original algorithm's reliance on a huge width per depth in the tree to explore more possible solutions. Since truncating length severely harmed the performance, I did not adopt it in later experiments, but since I haven't tested it on more expensive and slower models like GPT4, there's chance that it could be helpful.

4.3 Program-aided Evaluation

Inspired by the idea of incorporating a python runtime into the generating process (Gao et al., 2023) and using symbolic solver to evaluate the generation (Lyu et al., 2023) (Nye et al., 2021), I used a language model to generate a python script that can evaluate the model output symbolically given a customized prompt (shown in the Appendix). By using high-end models like GPT-4 to set an evaluator once a task is set in a runtime, the evaluation process can be done deterministically with up to 20 times faster speed, since it eliminated the time spent on prompting a language model to generate evaluations. Since we only need to design the prompt for the evaluator and it is set to be an attribute of a task, the original pipeline is still intact, but the cost efficiency, performance, and time consumption were all improved.

Table 2 shows that when symbolic evaluator was used, there are chances that smaller models would outperform larger models (Claude-3.5-Sonnet vs. GPT-4), and these experiments would not be possible in the first place if it were not for the symbolic solver. In Figure 1, we can see that the off-the-shelf ToT pipeline is extremely time-consuming, and since the tree could branch at exponential speed, the evaluation process became the bottleneck of the whole pipeline. The program-aided evaluator effectively solved this problem with the price of having to design a prompt that describes the task and evaluation criterion clearly.

4.4 Conclusion

In this project, I attempted to replicate some results from the original paper with limited resources. The effectiveness of the Tree-of-Thought (ToT) prompting technique seems to depend heavily on the language model used. I added three new features to the experimental setup. System prompts improved the performance of two baseline methods, making them more comparable to the ToT results in the

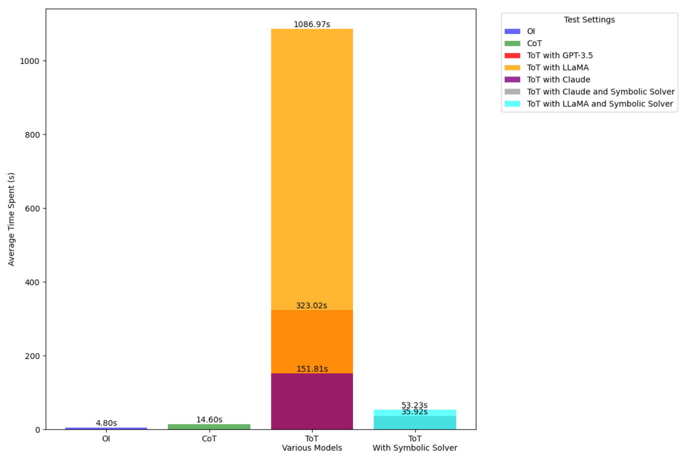

Figure 1: Time Consumption Comparison between different methods and models

original study. However, these prompts did not work well with the ToT pipeline and caused it to fail. This suggests that the ToT pipeline is not very robust. To stay within budget, I used techniques like truncating sequence length and early stopping, which affected how models and settings interacted. Adding a program-aided symbolic solver improved inference speed and success rates for some models.

In conclusion, the original ToT pipeline is not consistent across different models and tasks and is difficult to apply to various tasks. Although it's an interesting approach, it adds unnecessary complexity, reducing cost-effectiveness. My modifications helped improve the efficiency and quality of the Game24 task, but relying heavily on customized methods for specific tasks might not be the best approach.

References

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. 2023. Pal: Program-aided language models. In *International Conference on Machine Learning*, pages 10764–10799. PMLR.

Qing Lyu, Shreya Havaldar, Adam Stein, Li Zhang, Delip Rao, Eric Wong, Marianna Apidianaki, and Chris Callison-Burch. 2023. Faithful chain-of-thought reasoning. *arXiv preprint arXiv:2301.13379*.

Maxwell Nye, Michael Tessler, Josh Tenenbaum, and Brenden M Lake. 2021. Improving coherence and consistency in neural sequence models with dual-system, neuro-symbolic reasoning. *Advances in Neural Information Processing Systems*, 34:25192–25204.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. 2024. Tree of thoughts: Deliberate problem solving with large language models. *Advances in Neural Information Processing Systems*, 36.

A Appendix

A.1 System prompt

"You are a highly intelligent machine that follows the instructions. We are going to play a 24 puzzle, and a few examples will be provided, please do not add any notes to clarify your output or make your output too verbose, just stick to the format in the examples."

A.2 Evaluation prompt

"You are a highly reasonable and intelligent machine that follows the instructions. The task you are about to handle is using basic arithmetic operations to evaluate whether some numbers can reach 24, and a few examples will be provided. Please do not add any notes to clarify your output besides the thought process in the examples, just strictly stick to the format in the examples."

A.3 Evaluator generation prompt

*"Please generate a Python script that can solve 24 puzzles (using numbers and basic arithmetic operations (+ - * /) to obtain 24) with an command-line input with type list[int], which looks like this: [2, 4, 10], and the length would not exceed 4. The output should be a value indicate how likely this puzzle is solvable. So if it can be solved you should return 100, if not, look up the range the final results, if there is a number that's in the range of 10-40, then return 50, return 0 otherwise. Please make sure you only generate a executable Python code with a main function to take input and nothing else. Not a single word other than codes."*