Tree of Code: An LLM-Based Code Optimizer

Kohen Behar, Eran **Liu, Joy**
UC Berkeley CS285 Final Project
https://github.com/erankohen/cs285-project

Extended Abstract: In our project, we use a Tree-of-Thoughts approach to optimize Python code. We discuss how to assess and improve the code generation and code evaluation in a modular fashion. *Our contributions include (1) a grounded state evaluation method, (2) using different recommendations to prompt different branches of thoughts, and (3) a 214% increase in performance from the previous baseline.*

Research has shown that LLMs are capable of understanding and optimizing Python code. Shypula et al., 2023 created Performance-Improving Edits (PIE), a dataset of slow and fast pairs of Python code created by human programmers. Madaan et al., 2023 uses an iterative approach to have an LLM explain why the code is slow and then improve it given that feedback repeatedly. This improves the percent of code optimized in the PIE dataset to 21%. Our Tree-of-Thoughts approach to the problem is able to optimize 45% of the dataset, a 214% improvement on Madaan et al., 2023.

By conducting extensive failure analysis on the original PIE results, we realized that the model often failed when optimizing the code relied on more complex ideas or more creative thinking. Additionally, of the 21 the unique submissions that improved *at some point* of Self-Refine, only 10 remained improved by the final iteration.

Based on these insights, we applied Tree-of-Thoughts (Yao et al., 2023) to LLM-based code optimization, using external code execution as ground truth code evaluation as well as recommendations to diversify the action space. We call this approach Tree of Code (ToC). In the style of Tree of Thoughts, we break down the analysis of our design choices into four modular sections:

1. **Thought decomposition**: by grouping code feedback and code generation as intermediate reasoning steps of the same thought, we elegantly reformulate the state-action space representation for code generation.
2. **Thought generation**: based on our failure analysis, we use a series of recommendations to guide our actions. We provide analysis on how these recommendations affected the model.
3. **Thought evaluation**: we used three evaluation methods, predicting the improvement fraction, voting between candidates, and running on test cases. We find that running on test cases provides the best learning signal and run an ablation study on the number of test cases needed.
4. **Search algorithm**: we add skip connections to the BFS-ToT search algorithm that allow the model to reconsider the slow code.

1. Introduction

Worldwide, software engineers are afflicted by inefficient code. Large language models have proven themselves in a variety of fields, from artistic tasks like creative writing to mathematical reasoning. Improving code is a natural extension of existing applications of LLMs.

Planning processes that simultaneously explore many different branches of thinking are a fundamental characteristic of human problem-solving, as is evaluation of each branch to determine whether it is worth exploiting. To formalize this idea, we build on existing approaches of iterative refinement by giving the model different options at each phase. We augment the diversity of the state space by prompting the model with different ideas for each branch. We explore different code evaluation methods to reinforce the generation process both by running the intermediate code, and by using LLMs for evaluation as well.

2. Background

2.1. Performance Improving Code Edits

To test LLMs' performance on optimizing code, we use the Python dataset provided by the first version of "Learning Performance-Improving Code Edits" (Shypula et al., 2023). This dataset (called Performance Improving Edits, or PIE) contains 1000 pairs of slow and fast code written by humans for coding competitions, along with the speedup measurements of the $f$ast versions over the slow. It also contains 100 test cases for each coding problem in the dataset. Although the experiments in the PIE paper optimize up to 45.3% of the dataset, they use LLMs finetuned on code generation. In this paper, we explore the possibility of obtaining similar results using generic, non-finetuned LLMs (namely, gpt-3.5-turbo-1106).

In this paper, we define improvement fractions as Shypula et al., 2023 define "Runtime Reductions": $(\frac{o-n}{o}) \cdot 100$ where o is the average runtime of the original code and n is the average runtime of the generated code. Speedup is defined as $\frac{o}{n}$.

2.2. Self-Refine as a Baseline

Using the PIE dataset, Madaan et al., 2023 also aims to optimize code with general purpose language models. Their approach, called self-refine, iteratively asks the language model to first describe why the original code is slow, and then make it faster. With this self-reinforced code generation method, most of the dataset is optimized within 4 iterations of the feedback loop, or never optimized at all. They find that this approach gives an 8.2% improvement over single-sample improvement attempts (no reinforcement) using gpt-3.5-turbo (from 14.8% of the dataset optimized to 23%).

We recreated the results of Madaan et al., 2023 to find very similar results. Due to resource constraints we ran our experiments on 10% of the dataset and found that 21% of that is optimized with the self-refine approach. We use this as a baseline for our novel experiments.

Our analysis showed that many entries of the dataset improve in the first one or two iterations of self-refine and stop improving afterwards. Of the 21 unique submissions that improved *at some point* of Self-Refine, only 10 remained improved by the final iteration. We aim to solve this problem with Tree of Code by allowing earlier versions of the code to be re-iterated on as well, as described later in section 3.4.

After looking through many cases where self-refine failed to optimize code, we found two common challenges that this approach faced: (1) some problems require mathematical reasoning which the language model does not perform, and (2) the model often failed to make large changes to the code in order to optimize it. These problems can be somewhat reduced by prompting the LLM to use mathematical reasoning, and to be more creative in changing the structure of the code, respectively.

For example, in the first box below we see an example input that requires mathematical reasoning to optimize, and the results given with the self-refine prompts. The model is not able to realize that the least common multiple of any number and 2 is either the number itself, or 2 times the number. Therefore, the feedback it gives claims the code is not slow, and the generated code is not very different.

However, in the next box, we follow the same process but add "using mathematical reasoning" to the prompt. This time, the model is able to realize the simpler approach to finding the LCM with 2, and generates much faster code.

Default prompt

Slow code:
```
import fractions
def lcm(x, y):
 return (x * y) /fractions.gcd(x, y)
N = int(eval(input()))
print(lcm(N, 2))
```

 # Why is this code slow?
```
"The code may seem to be slow because
of the `input()` and `eval()`
functions, which are used to get and
process the user's input. They can be
quite slow for large inputs. However,
the `lcm(N, 2)` computation is quite
fast."
```

 # Improved version:
```
import fractions
N = int(input())
print((N * 2) //math.gcd(N, 2))
```

Mathematical reasoning prompt

Slow code:
```
import fractions
def lcm(x, y):
 return (x * y) /fractions.gcd(x, y)
N = int(eval(input()))
print(lcm(N, 2))
```

 # Using mathematical reasoning, explain why this code is slow.
```
"The code calculates the least common
multiple (LCM) of N and 2. The Python
fractions.gcd() function uses the
Euclidean algorithm, which has a
worst-case time complexity of O(log
min(a, b)) where a and b are the two
input numbers.
In this specific instance, the input
to the gcd function is always 2 and
N. The gcd of 2 and N is always 2 if N
is divisible by 2, and 1 otherwise.
Therefore, the time complexity of
gcd(2, N) is constant: O(1)."
```

 # Improved version
```
def lcm(N):
 return N if N % 2 == 0 else N * 2
N = int(input())
print(lcm(N))
```

2.3. Tree-of-Thoughts

The Tree of Thoughts paper (Yao et al., 2023) builds on the Chain of Thought idea proposed by Wei et al., 2023. Chain of Thought demonstrates that prompting a language model to take a step-by-step approach to problems helps it decompose the problem into smaller parts and solve it more accurately. To get more robust results, they prompt the LLM multiple times and choose the ”best” result. Yao et al., 2023 take this approach one step further by sampling at the ”thought” level instead of at the problem level. The Tree-of-Thoughts (ToT) approach asks an LLM to generate small steps (or thoughts) toward the solution of a larger problem. Then, it chooses a few of the ”best” thoughts, and asks the LLM to generate the next step based on these to obtain a new layer of thoughts to choose from. Applying this iteratively, ToT manages to get very promising results on three tasks (creative writing, a mathematical game of 24, and crosswords).

Yao et al., 2023 decompose the ToT approach into 4 main sections: (1) thought decomposition, (2) the thought generator, (3) the state evaluator, and (4) the search algorithm. We describe how we use and change each of these parts of the algorithm in Section 3, Methods.

The self-refine approach to the code optimization process is akin to Chain of Thought in its breaking down of the problem into sequential steps. In this paper, we investigate whether a Tree of Thoughts-like approach can perform better in this context as it does in other problems.

3. Methods

We made several design choices when determining how to navigate the state space of code in ToC.

3.1. Thought decomposition

Thoughts have to be small enough that LLMs can generate diverse samples, but big enough that LLMs can evaluate how promising they are. While Self-Refine (Madaan et al., 2023) separated the generation of code explanation and code snippets, we follow the lead of Chain of Thought (Wei et al., 2023) in which thoughts are sampled coherently without need for explicit decomposition. In the same API call, we prompt the model to explain why the code is slow, and generate a more optimized version of the code. Our justification for this is twofold. Firstly, we determined that given the same explanation, LLMs did not yield diverse code generations, suggesting that improvements to the code follow naturally from an explanation, and code and explanation together constitute one thought. Additionally, the code generation is still conditioned on the feedback within the same response, and so we expect that separating the prompts has no significant benefit.

3.2. Thought generator

Tree of thought works best when there is a diverse observation space. From our failure analysis of Self-Refine, we learned that the code could benefit from different prompts.

For each snippet of slow code, we had a branching factor of three, where we asked the model to improve the code:

1. ”using mathematical reasoning where you use critical thinking and logical thinking to solve mathematical problems”,
2. ”using creative solutions where you rethink the structure of the code”,
3. or without any specific recommendation.

To see how these recommendations are added to the LLM prompt, please refer to our GitHub repository.

3.3. Thought evaluator

We used three different state evaluation methods: vote, value, and oracle. The vote and value evaluations are the same as described in the Tree of Thoughts paper: Value assignment prompts the LLM to consider generated states independently and return a scalar value for each state, while voting gives the LLM all candidates and prompts it to choose the best one. In our case, for value evaluations, we prompted the LLM to predict the improvement fraction of the code with a few-shot prompt, and the voting was given all generated samples in a zero-shot prompt.

Self-Debug (Chen et al., 2023) showed that it can be helpful to investigate the execution results at intermediate stages. Following their lead, we adapted the PIE codebase to create an oracle value function grounded in true accuracy and improvement fractions of generated code based on input-output pairs. In the PIE dataset, each code submission has 100 test cases for evaluation. Using these tests makes our state value estimation much more accurate, allowing for improved code optimization.

We hypothesize that test-based evaluation will perform significantly better than LLM-based vote or value functions. This is because we do not expect the latter two to be accurate heuristics, and reinforcing the code generation process with them reduces the learning signal. We confirm this hypothesis in Section 4.1

3.4. Search Algorithm

Motivated by the possibility that the code does not improve during an iteration, we make a modification to the ToT-BFS search algorithm that allows the model to revisit the slow code. Instead of choosing the best two candidates out of the newly generated 6 code snippets at each layer, we also consider the two selected nodes from the previous layer. This way, if attempting to make the code faster makes it inaccurate in all branches, we can choose to re-iterate on the same snippet again until we reach the depth limit.

4. Experiments

In this section, we assess the diversity of our sample generation and the accuracy of our sample evaluation process. We ran all our experiments with the gpt-3.5-turbo-1106 language model on MacBook Pros. It is possible that evaluation times were measured incorrectly due to other tasks being scheduled on the same device. However, this should not cause very large differences in our analysis, especially because each measurement is taken multiple times. To reduce such errors, we also check p-values in speedup calculations and ignore any speedup with a p-value greater than 0.05.

Unless otherwise specified, all experiments are conducted on the first 100 code snippets. We verified that the order was random by comparing improvement fractions for the human reference code.

Our first experiment was to re-run self-refine (Madaan et al., 2023) as a baseline. We found that 21% of the dataset was optimized.

	Self-Refine	ToC with Vote	ToC with Diverse Prompts and Oracle	ToC with Oracle
% OPT	21	14	32	45
Speedup	6.65 ± 3.54	9.13 ± 5.07	7.48 ± 5.62	3.01 ± 2.06

Table 1: Comparing percent of code optimized and speedup for various runs.

4.1. Thought evaluator

In this section, we detail the experiments used to compare different evaluation methods. We prompted the LLM to predict the improvement fraction of each code snippet and to vote between all code snippets. For the value method, we averaged the predicted improvement fraction over three API calls. For the voting method, we made three API calls containing every code snippet generated. Simultaneously, we calculated test-based (oracle) improvement fractions on all 100 input-output pairs. If the accuracy was below 1.0, we replaced this improvement fraction with $-\infty$.

To assess the evaluation methods, we ran ToT on 50 slow code snippets. We sorted the list of oracle values in decreasing order and extracted the indices of the two candidates that were greedily selected with the value or vote methods. We note that the smaller index of the two selected candidates tends to be quite promising, whereas the larger is almost random, especially for the value method (as seen in Figure 1).

Having determined that voting yields more promising candidates, we ran ToT on 100 slow code snippets using the voting method and also the oracle method. As Table 1 shows, the test-based oracle evaluation largely outperformed the voting method.

However, while the results with oracle evaluation are very good, running on 100 input-output pairs for every node is computationally intensive, and generating test cases is difficult for programmers. We hypothesized that the model would perform similarly well with fewer test cases, and performed an **ablation study** on the number of tests used during state evaluation. In Table 2 we see that performance improvement slows as we increase the number of test cases used, indicating that the programmer can predictably trade number of tests for performance.

Num test cases	**1**	**5**	**10**	**100**
% OPT	31	30	37	45

Table 2: Ablation on number of test cases. We see that increasing the number of tests used for grounded state evaluation has diminishing returns in terms of the percent of code optimized.

4.2. Thought generator

Next, we test adding recommendation to the prompt. We add recommendations to the prompt for two of the three samples from each node. As described in section 3.2, the first sample has no recommendations, the second has a recommendation to use mathematical reasoning, and the third to be more creative.

After adding the recommendations, we found that the percent of data optimized was reduced by 28.9%, from 45% to 32% when using our oracle value function with all 100 tests.

As hypothesized, using diverse prompts leads to more variety in the samples. A measure for this is the variance in performance of the generated code. Table 3 shows the average standard deviation of the improvement fractions within each iteration of the Tree of Thoughts algorithm. We see that with diverse prompts, we have higher standard deviation, and presumably more varied code. However, in Table 3, we also see that the average of the maximum improvement fractions at each layer are better by an order of magnitude with uniform prompts. This is because although the recommendations are helpful to get more variance in improvement attempts, our prompts were not conducive to the model writing better code.

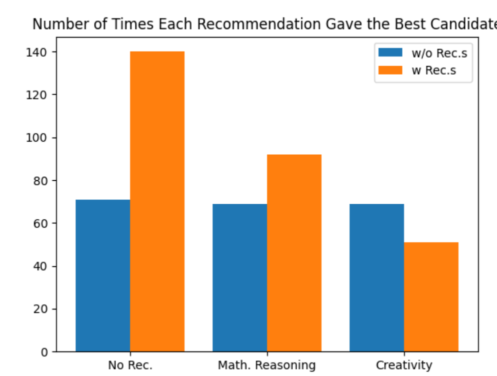

Figure 2: The number of times each recommendation resulted in the best candidate at any layer of any problem. The blue bars are for uniform prompts, when recommendations are not used. Since the prompts in all three branches are the same, all three indices are equally likely to be selected.

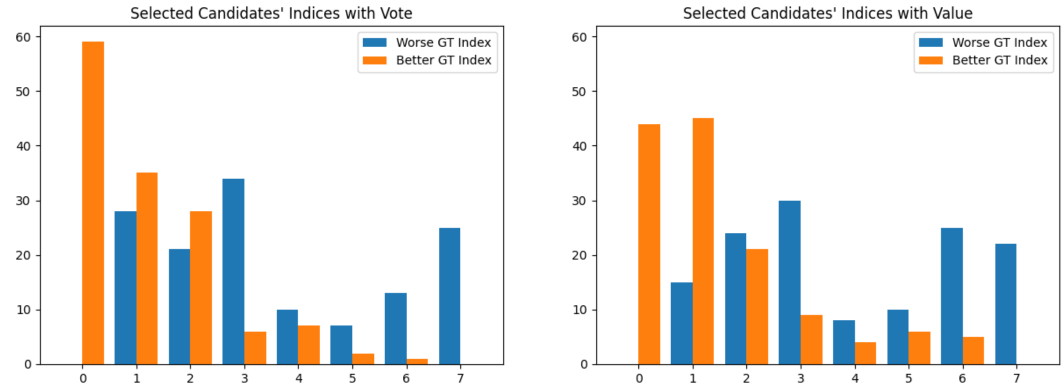

Figure 1: The oracle indices of candidates selected by Vote (left) and Value (right). Lower indices are better performing candidates. Of the two selected candidates in each layer, the better one (orange) tends to have lower oracle indices (almost 60% for Vote). The worse candidate that is selected (blue) does not tend to be a good option. We also see that Vote is much more likely to select the best two candidates, and is therefore a better evaluation method.

ToT Levels	**0**	**1**	**2**	**3**
Diverse Prompts	-136.59 ± 2.12	-93.45 ± 0.68	-84.00 ± 0.38	-82.52 ± 3.90
Uniform Prompts	-3.55 ± 8.08	-5.29 ± 5.21	-5.70 ± 0.89	-7.58 ± 1.71

Table 3: Mean and Standard Deviation of improvement fractions with and without recommendations at each layer of tree-of-thoughts. The improvement fractions of inaccurate code were ignored in these calculations. Any layer with less than or equal to 1 accurate node were also ignored. The standard deviations are calculated for each layer of each problem, and the average of these are displayed.

To get a better understanding of how the different recommendations performed, we also break down how many times the result of each prompt was selected as the ”best” within a layer. This is displayed in Graph 2. We see that the result from the ”no recommendation” sample is preferred significantly more than either of the recommendations. Therefore, we infer that our recommendations do not help generate better code in general, even though they are helpful in some specific scenarios.

Our overall results can be found in Table 1. We find that to optimize the largest number of code snippets, ToC with uniform prompts and oracle evaluation is best. However, there is also a trade-off between the number of problems optimized and the amount they are sped up by.

5. Future Research

Future research can rely on the metrics we've introduced to improve thought evaluation and generation in a modular fashion. To improve evaluation further, we can first of all try having more votes to see if the second selection is improved. If not, we can take advantage of the fact that voting and is rather good at selecting the best option, even if the second is rather random. A new evaluation method where we ask the language model to vote on the best candidate, take the option with the most votes out, and then re-vote on the smaller set might help choose the second best option more accurately.

To improve generation, we believe recommendations are promising and merits further investigation. As our experiments showed that irrelevant recommendations can harm performance, yet failure analysis suggests recommendations could be crucial in exploring the full state space of code snippets. We would suggest having a wider array of possible recommendations, or a higher branching factor, to increase diversity of options. There should exist a tradeoff between exploring different recommendations broadly and exploiting specific recommendations.

6. Conclusion

In this paper, we optimize Python code using Tree of Code, a deep reinforcement learning algorithm using a Tree of Thought framework. In particular, we are able to improve a larger percent of programs in the PIE dataset. Our results are an interesting step toward capturing the thinking process of writing good code.

References

Chen, X., Lin, M., Schärli, N., & Zhou, D. (2023). Teaching large language models to self-debug.

Madaan, A., Tandon, N., Gupta, P., Hallinan, S., Gao, L., Wiegreffe, S., Alon, U., Dziri, N., Prabhumoye, S., Yang, Y., Gupta, S., Majumder, B. P., Hermann, K., Welleck, S., Yazdanbakhsh, A., & Clark, P. (2023). Self-refine: Iterative refinement with self-feedback.

Shypula, A., Madaan, A., Zeng, Y., Alon, U., Gardner, J., Hashemi, M., Neubig, G., Ranganathan, P., Bastani, O., & Yazdanbakhsh, A. (2023). Learning performance-improving code edits.

Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., Chi, E., Le, Q., & Zhou, D. (2023). Chain-of-thought prompting elicits reasoning in large language models.

Yao, S., Yu, D., Zhao, J., Shafran, I., Griffiths, T. L., Cao, Y., & Narasimhan, K. (2023). Tree of thoughts: Deliberate problem solving with large language models.