Enhancing Large Language Model's Coding Ability by Tree-Based Searching Methods

Heyang Yu†
Tsinghua University
Haidian, Beijing, China
hy-yu22@mails.tsinghua.edu.cn

Xiangyu Zhang†
Tsinghua University
Haidian, Beijing, China
xiangyuz22@mails.tsinghua.edu.cn

Ziheng Zhou†
Tsinghua University
Haidian, Beijing, China
zhouzh22@mails.tsinghua.edu.cn

Abstract

Code generation remains a pivotal challenge in machine learning, requiring models to generate accurate code based on problem descriptions. This study investigates reinforcement learning methods for code generation using a dataset of 76 programming problems and their corresponding test cases. We explore four approaches: the baseline Best-of-N (*Vanilla*), *MCTS-Append*, *MCTS-Modify*, and *Tree of Thought (ToT)*.

Our results demonstrate that *Tree of Thought* outperforms other methods on simpler problems, achieving the highest pass rate and lowest computational budget when full test case feedback is available. However, its performance declines on more complex tasks. MCTS-based methods show limited improvements over the baseline, often hindered by suboptimal early reasoning paths and potential prompt poisoning effects, wherein previously generated code adversely impacts model outputs. When evaluated with partial test case feedback, *MCTS-Append* and *Tree of Thought* demonstrate improved validation set performance, highlighting their potential in realistic scenarios where complete feedback is unavailable.

These findings underscore the challenges of adapting tree-based reasoning for code generation and the importance of robust prompt design. Future work should address these limitations by refining initial reasoning strategies, mitigating prompt poisoning, and exploring hybrid methods to enhance code generation in large language models.

Keywords

Reinforcement Learning, LLM, Code Generating

1 Introduction

Code generation is a significant problem in the field of machine learning. In the code generation task, given a problem description, the goal is to generate code that solves the problem. In this project, we aim to address this problem using reinforcement learning methods.

Specifically, in this task, we are given 76 programming problems along with their complete test cases (10 test cases per problem). The evaluation of a piece of code is determined by the number of test cases it passes. We are required to solve these problems using a provided base model, interacting with the model via API calls. For each problem, the model can be invoked up to N times, where N is a given budget limit. Our method outperforms the baseline on the first 40 relatively easier problems but underperforms on the remaining 26 more challenging ones.

†These authors contributed equally to this work.

2 Related Work

Shinn et al. [1] proposed a new framework named *Reflexion*, which enhances model performance through language feedback instead of updating model weights. Wei et al. [2] introduced the *Chain of Thought (CoT)* method, which improves the ability of large models to solve complex reasoning problems by generating a series of intermediate reasoning steps and forming them into a chain. Building on this, Yao et al. [4] proposed the *Tree of Thought (ToT)* method, which enables models to explore multiple reasoning paths and self-assess to determine the next step, and demonstrated its effectiveness in solving complex problems like the 24-game. Wu et al. [3] applied tree search algorithms to solve mathematical problems, achieving promising results.

3 Methodology

The simplest approach to this task is to use the problem description as a prompt and invoke the model N times, selecting the best output among the N generated codes. Given the availability of all test cases, we can accurately assess the quality of each generated code. We refer to this straightforward best-of-N approach as *Vanilla*.

While this method is simple, it is inefficient because subsequent model invocations do not leverage information from prior attempts. To improve efficiency, we aim to utilize the budget more effectively by incorporating information from previously generated codes into subsequent attempts. Specifically, we use the *Monte Carlo Tree Search (MCTS)* method, where each node in the tree stores a piece of code. In each iteration, we apply the *Upper Confidence Bound (UCB)* algorithm layer by layer to select a leaf node for expansion. During expansion, the problem description combined with the first dK lines of the current node's code is used as the prompt to guide the model in completing the code. Here, d is the depth of the node, and K is a constant. In other words, the prompt at each node consists of the problem description and a prefix of the parent node's code, with the prefix length increasing by K at each layer. After expansion, each node is evaluated using the test cases, and the results are back-propagated to update the value of all ancestor nodes. The value of a node is updated to the maximum of its original value and the current reward, representing the potential of the code prefix at that node to be completed into correct code. When the algorithm ends, i.e., when the budget is exhausted, the best-performing code in the tree is selected as the result.

Return the code at the node with the highest value This method is referred to as *MCTS-Append*. A variant of this approach, called *MCTS-Modify*, differs only in the prompts used. Instead of using the problem description and a prefix of the parent node's code,

Algorithm 1 MCTS-Append

Initialize root node with empty code C = ""
for iteration = 1 to ℓ **do**
  Select a leaf node using Upper Confidence Bound (UCB)
  Expand the selected node:
    Append K lines of code to the current node's code C
    Use P and the first dK lines of code as the prompt
    Generate a new code candidate
  Evaluate the candidate code with test cases
  Backpropagate the evaluation results to update ancestor nodes
end for
return the code at the node with the highest value

Algorithm 2 MCTS-Modify

Initialize root node with empty code C = ""
for iteration = 1 to ℓ **do**
  Select a leaf node using Upper Confidence Bound (UCB)
  Expand the selected node:
    Use the complete code from the parent node as the base
    Modify the code using P as the guiding prompt
    Generate a new code candidate
  Evaluate the candidate code with test cases
  Backpropagate the evaluation results to update ancestor nodes
end for
return the code at the node with the highest value

Algorithm 3 Tree of Thought (ToT)

Initialize root node with an empty thought T = ""
for iteration = 1 to ℓ **do**
  Select a thought node using Upper Confidence Bound (UCB)
  Expand the selected node:
    Refine the current thought T
    Use P and the refined thought as the prompt
    Generate a new thought and corresponding code
  Evaluate the generated code with test cases
  Backpropagate the evaluation results to update ancestor nodes
end for
return the code corresponding to the thought with the highest value

MCTS-Modify uses the problem description and the complete code of the parent node, prompting the model to modify the code. We believe this method could address issues where errors in the initial part of the code cannot be corrected.

Additionally, inspired by Yao et al. [4], we believe the *ToT* method can also be applied to the code generation task. Here, we maintain a tree structure similar to the one described above, but this time the model first generates a *writeup* for the problem, describing the algorithm, data structures, etc., without including the code. Each time a thought is generated, it is *rolled out* by using the problem description and the thought as a prompt for the model to generate code. The correctness of the code serves as the thought's value. During expansion, the problem description and the thought are used as prompts, instructing the model to refine and modify the thought before rolling it out again.

4 Experiment

4.1 Feedback from All Test Cases

In the first part of our experiment, the methods were evaluated by obtaining execution feedback from all 10 test cases for each question. We tested four methods: *Vanilla* (Best-of-N), *MCTS-Append*, *MCTS-Modify*, and *ToT(Tree of Thought)*. These methods were evaluated on a set of 40 simple questions from the dataset, with a maximum budget of 100 rollouts per question. For all tree-based methods, the expansion width was set to 3, and the *MCTS-Append* method appended 5 lines of code per iteration. The results are summarized in Table 1.

Method	Average Pass Rate	Average Budget
Vanilla (baseline)	67.4%	47.825
MCTS-Append	65.1%	48.475
MCTS-Modify	62.2%	49.325
ToT	**68.3%**	**45.975**

Table 1: Performance of methods with feedback from all test cases.

The results indicate that the *Tree of Thought* method outperformed the other three methods on the set of 40 simple questions, achieving both the highest pass rate and the lowest budget. However, MCTS-based methods did not surpass the baseline performance. We hypothesize that incorrect code suggestions provided to LLM may be adversely affecting its performance, effectively poisoning the model. To investigate this hypothesis, we included previously generated code in the prompts of the Vanilla method and evaluated it on the same set of 40 simple questions, with a reduced maximum budget of 30 rollouts per question. The results are presented in Table 2.

Method	Average Pass Rate	Average Budget
Vanilla w/o code	**60.3%**	19.125
Vanilla w/ code	42.0%	22.375

Table 2: Effect of including previously generated code in prompts.

The results support our hypothesis, suggesting that previously generated code negatively impacts the LLM's performance. Subsequently, we tested all the methods on the full dataset, consisting of 76 questions, including 36 more challenging ones. The maximum budget remained at 100 rollouts per question, with the same parameter settings for tree-based methods. The results are shown in Table 3.

None of the tree-based methods outperformed the baseline on the full dataset, particularly when tackling more challenging questions. Combined with the hypothesis of prompt poisoning, we conclude

Method	Average Pass Rate	Average Budget
Vanilla (baseline)	**64.1%**	50.947
MCTS-Append	62.9%	50.250
MCTS-Modify	58.5%	53.605
ToT	57.8%	53.921

Table 3: Performance of methods on the full dataset.

that tree-based methods may struggle to generate correct code or reasoning paths at the early stages of the search process, resulting in suboptimal downstream performance. Furthermore, granting access to all test cases provides an advantage to the Best-of-N method, as it consistently selects the best performing outputs across both test and validation sets.

4.2 Feedback from Partial Test Cases

In the second part of our experiment, we provided only half of the test cases to each method, reserving the remaining half for validation. This setup reflects a more realistic evaluation scenario than in the first part. All methods were tested in the entire dataset, with a maximum budget of 100 rollouts per question. The parameters for the tree-based methods remained unchanged. The results are presented in Table 4.

Method	Test Set	Valid Set	Average Budget
Vanilla (baseline)	60.3%	63.5%	48.658
MCTS-Append	54.7%	**65.5%**	50.447
MCTS-Modify	55.0%	60.4%	53.474
ToT	58.6%	64.9%	49.592

Table 4: Performance of methods with feedback from partial test cases.

The results demonstrate that both the *MCTS-Append* and *Tree of Thought* methods performed better than the baseline in the validation set, although the baseline achieving the highest pass rate on the test set. This reinforces our conclusion that access to all test cases is inherently biased in favor of the baseline method. Consequently, the effectiveness of the *MCTS-Append* and *Tree of Thought* methods is validated under this more balanced evaluation framework.

5 Conclusion

In this study, we evaluated four methods, Vanilla (Best of N), *MCTS-Append*, *MCTS-Modify*, and *Tree of Thought*, under different experimental setups to assess their effectiveness in solving programming problems using large language models (LLMs). The experiments were carried out in two scenarios: access to all test cases and access to only partial test cases, reflecting different evaluation conditions.

Our findings highlight the following key points:

- The *Tree of Thought* method demonstrated superior performance on simple questions with access to all test cases, achieving the highest pass rate and the lowest average budget. However, it underperformed on the more challenging questions in the full dataset.
- MCTS-based methods (*MCTS-Append* and *MCTS-Modify*) struggled to outperform the baseline in most scenarios, particularly when dealing with difficult questions. This limitation may be attributed to their inability to generate effective initial reasoning paths or code during the early stages of the search process.
- The hypothesis of prompt poisoning was validated, as the inclusion of previously generated code in the Vanilla method's prompts significantly degraded its performance. This underscores the importance of careful prompt design to avoid unintended adverse effects on LLM.
- When evaluated with partial test case feedback, the *MCTS-Append* and *Tree of Thought* methods showed improved performance on validation sets compared to the baseline, suggesting their potential effectiveness in realistic scenarios where complete test case feedback is unavailable.

Overall, while the *Tree of Thought* and *MCTS-Append* methods demonstrated promise under certain conditions, their performance was inconsistent, particularly for complex problems. Future work should focus on addressing the limitations of tree-based methods by improving their ability to generate robust initial reasoning paths and mitigating the effects of prompt poisoning. Furthermore, exploring hybrid approaches that combine the strengths of different methods could further enhance their applicability to solving programming problems using LLM.

References

[1] Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language Agents with Verbal Reinforcement Learning. arXiv:2303.11366 [cs.AI] https://arxiv.org/abs/2303.11366

[2] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2023. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. arXiv:2201.11903 [cs.CL] https://arxiv.org/abs/2201.11903

[3] Yangzhen Wu, Zhiqing Sun, Shanda Li, Sean Welleck, and Yiming Yang. 2024. Inference Scaling Laws: An Empirical Analysis of Compute-Optimal Inference for Problem-Solving with Language Models. arXiv:2408.00724 [cs.AI] https://arxiv.org/abs/2408.00724

[4] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. Tree of Thoughts: Deliberate Problem Solving with Large Language Models. arXiv:2305.10601 [cs.CL] https://arxiv.org/abs/2305.10601