CodePRM: Execution Feedback-enhanced Process Reward Model for Code Generation

Qingyao Li[1,2], **Xinyi Dai**[2], **Xiangyang Li**[2], **Weinan Zhang**[1*], **Yasheng Wang**[2], **Ruiming Tang**[2], **Yong Yu**[1*],
[1]Shanghai Jiao Tong University, [2]Huawei Noah's Ark Lab
{ly890306,wnzhang}@sjtu.edu.cn, yyu@apex.sjtu.edu.cn
{daixinyi5, lixiangyang34, wangyasheng, tangruiming}@huawei.com

Abstract

Code generation is a critical reasoning task for large language models (LLMs). Recent advancements have focused on optimizing the thought process of code generation, achieving significant improvements. However, such thought process lacks effective process supervision, making it hard to optimize the thoughts. Although Process Reward Models (PRMs) have been widely established in mathematical reasoning, building a code PRM is still not trivial for the gap between thoughts to code. In this paper, we propose CodePRM, a novel approach that leverages the code execution feedback to build a code PRM. Specifically, we first collect a large dataset of thought traces, where each thought step is labeled with their derived code' pass rates, accompanied by the corresponding code snippets, and execution feedback. During training, we train a PRM to take both the reasoning process and code execution feedback as input to score individual thought steps, enabling it to leverage code execution results to distinguish between high-quality and low-quality thought steps. Finally, to use the PRM during inference, we develop a Generate-Verify-Refine (GVR) pipeline where the CodePRM serves as a process verifier to dynamically identify and correct errors in the thought process during code search. Experimental results demonstrate that CodePRM with the inference algorithm outperforms strong baselines, significantly enhancing code generation performance. Further analysis reveals the key factors for building a code PRM.

1 Introduction

Code generation is a task that demands advanced reasoning capabilities (Fu et al., 2023; Le et al., 2024; Ni et al., 2024). Early efforts to enhance LLMs' performance in code generation primarily focused on direct optimizations at the code level (Le et al., 2023; Chen et al., 2024), such as

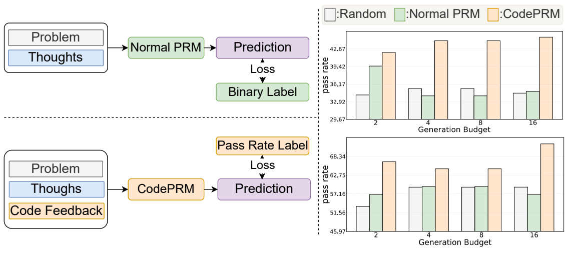


Figure 1: The training paradigms and performance comparison of normal PRM and CodePRM. Without code and execution feedback-enhancement, code PRM would perform no better than random reward on Best-of-N algorithm.

decomposing code (Shi et al., 2024) or modifying code based on variable values change (Zhong et al., 2024). Recently, however, there has been a shift in focus toward improving the underlying thought processes that drive code generation. Numerous tree search algorithms have been proposed to explore and refine these thought processes (Li et al., 2024a; Wang et al., 2024a; Chen et al., 2023; Li et al., 2024b), demonstrating significant potential and achieving promising results.

Despite these advancements, a significant gap remains in developing effective process supervision for the thought processes involved in code generation. Unlike mathematical reasoning, where process reward models (PRMs) have been extensively studied, research on PRMs for code generation is limited. This gap stems from two key challenges: (1) Defining Intermediate Steps: In code generation, the granularity of a "step" lacks a consistent definition. A step could be an element in the code space (e.g., a line of code (Wang et al., 2024d; Dai et al., 2024) or a token of code (Zhang et al., 2023)) or a natural language-based reasoning step (Li et al., 2024b; Wang et al., 2024a). Recent studies suggest that focusing on thought steps is more effective for LLMs in code generation (Li et al., 2024b,a; Wang et al., 2024a). (2) Mapping Code Errors to Thought Processes: In mathematics,

*Corresponding authors.

intermediate steps often contain partial computation results, making it easier to trace errors in the final answer back to middle thoughts. While in code generation, the thoughts are usually abstract and involve algorithmic design, workflow construction, and data structure definition, making it harder to pinpoint errors in the thought process that lead to faulty code. Empirically, as illustrated in Figure 1, we find that training a code PRM using the same paradigm as mathematical PRMs performs no better than random reward assignment, demonstrating that developing an effective code PRM requires innovative paradigms tailored to the unique challenges of code generation.

This paper presents a new perspective on the problem. The idea is that thought processes in code generation should not be evaluated in isolation; instead, they should be assessed in conjunction with the generated code and execution feedback. The feedback in the code environment usually contains detailed error information about the code, which would be helpful in identifying process errors.

In light of this, we propose leveraging the code and execution feedback as auxiliary information to construct a robust CODEPRM. Specifically, we first employ tree search algorithms to generate and collect a dataset of labeled thought steps for code generation. Each thought step is annotated with the average pass rates of the code derived from it. During this process, we also store the corresponding code snippets and execution feedback associated with each thought. Subsequently, we train a vanilla LLM into a feedback-enhanced CODEPRM through supervised fine-tuning. CODEPRM incorporates both the code and execution feedback as auxiliary inputs when scoring thought steps. As illustrated in Figure 1, this new paradigm delivers significant performance improvements.

To apply CODEPRM in inference, we design a Generate-Verify-Refine (GVR) framework. In this framework, a generator LLM explores diverse thought steps and corresponding code using tree search algorithms. CODEPRM then acts as a verifier to identify erroneous thought steps. The erroneous thoughts would be identified and then refined by the generator LLM, which ultimately leads to improved code. Overall, our key contributions are as follows:

- We construct an effective feedback-enhanced CODEPRM. To the best of our knowledge, we are the first to systematically study the process reward model for code generation.
- Based on CODEPRM, we develop the Generate-Verify-Refine (GVR) inference pipeline that integrates the CODEPRM to dynamically identify and correct errors in the reasoning process during code generation.
- We conducted comprehensive experiments to validate the effectiveness of CODEPRM and have made the resources and model weights of the trained CODEPRM available during the review process[1]. It would fully open-sourced upon publication, which we believe will serve as a foundational resource and provide valuable insights for future research on code process supervision.

2 Related Work

2.1 Large Language Models for Code Generation

In code generation, early research has predominantly concentrated on algorithm design or data collection and fine-tuning at the code level (Shen et al., 2023; Luo et al., 2023; Shi et al., 2024; Zhong et al., 2024). LATS (Zhou et al., 2023) employs the Monte Carlo Tree Search (MCTS) algorithm to iteratively search through code and incorporates a reflection mechanism to learn from past mistakes, thereby improving search quality. Since the release of OpenAI's o1 model (OpenAI, 2024b), there has been a growing emphasis on the "thought process" behind code and mathematics, especially in the form of long chain-of-thought (CoT) (Wang et al., 2024a; Jiang et al., 2024; Li et al., 2024a). For example, PlanSearch (Wang et al., 2024a) explores coding ideas and strategies using tree search algorithms, enhancing the diversity of the generated code. RethinkMCTS (Li et al., 2024b) introduces a refinement mechanism into the code search process, leveraging execution feedback to correct and refine the search direction during the search. Although these approaches have yielded promising results, there has been limited exploration into process reward models (PRMs) for supervising the thought steps in code generation. This paper aims to investigate and develop an effective code PRM, providing process supervision for code search and reasoning algorithms.

[1]The resources of this work are made available at https://github.com/SIMONLQY/CodePRM.

2.2 Process Reward Model

The Process Reward Model (PRM) enhances reasoning quality by providing fine-grained reward signals for intermediate steps in generation processes (Setlur et al., 2024; Zhang et al., 2025), guiding models to optimize reasoning trajectories. Previous research on PRMs has predominantly focused on mathematical tasks (Wang et al., 2024d,b; Liang et al., 2024). Due to the strong correlation between the middle thought steps and the final answer in mathematics, PRMs have shown significant performance (Luo et al., 2024; Wang et al., 2024c,d; Xiong et al., 2024; Lu et al., 2024). In contrast, constructing code PRMs is a challenging task. Few works have investigated this area. Yu et al. (2024) utilize code outcomes as inputs to prompt chat LLMs in providing process rewards, while our work focuses on training a specialized code PRM and utilizing it as a verifier for thought refinement. While these studies have thoroughly examined the construction of PRMs, there is a lack of systematic discussion and research on training and applying a code PRM. This paper aims to fill this gap and provide a foundation for future research on code PRMs.

3 Method

The workflow of our method is illustrated in Figure 2. First, we collect intermediate thought steps data for code, where each thought is annotated with the average pass rate of the code it subsequently derives. With the code and execution feedback as an auxiliary input, we fine-tune an LLM through supervised fine-tuning to develop a feedback-enhanced PRM. During inference, we propose a Generate-Verify-Refine (GVR) pipeline, which can be integrated with various search algorithms. In this pipeline,CodePRM serves as a thought verifier, identifying incorrect thoughts during the search. These erroneous thoughts are then refined to produce higher-quality code.

3.1 Data Collection

To build CodePRM, the first and foremost is to collect labeled thought steps and corresponding code data. Similar to the automated PRM data collection strategy used in mathematics (Luo et al., 2024), we adapt a tree search procedure to systematically explore possible thought steps and the associated code.

Tree Search for Thoughts and Code Given a problem description x, we begin by creating a root node s_0 in the tree search, whose state contains only the problem text. From this root node, we expand child nodes by generating thought steps $(t_1, t_2, ...)$, and each node's state is the concatenation of its parent node's state with the new thought step $s_i = (x, t_1, t_2, ..., t_i)$. The tree grows as we repeatedly sample and add these thought steps, incrementally constructing chains of reasoning. At various stages (e.g., after a few thought steps), the system completes the chains and generates the corresponding code c following the thoughts.

Execution Feedback and Pass Rate The generated code is executed on test cases, producing two scalar feedback values: $p_{\text{pass}}^{\text{pub}}$ as the pass rate on public test cases and $p_{\text{pass}}^{\text{priv}}$ as the pass rate on private test cases. Additionally, verbal feedback f is provided, detailing error information for any failed test cases. Specifically,

$$f = \begin{cases} \text{All public test cases passed}, & \text{if } p_{\text{pass}}^{\text{pub}} < 1 \\ \{\text{Error Information}\}, & \text{if } p_{\text{pass}}^{\text{pub}} = 1. \end{cases} \tag{1}$$

Each node i would finally contain a tuple $(s_i, p_{\text{pass}}^{\text{pub}}, p_{\text{pass}}^{\text{priv}}, c, f)$ referring to the node state, public pass rate, private pass rate, code and execution feedback.

Labeling Thought Steps by Average Pass Rate One feature of our data collection approach is to label each intermediate thought step with an average pass rate of all the code it derives. The $p_{\text{pass}}^{\text{priv}}$ of each node represents the label and would be updated with the tree growth. Specifically, whenever we complete and execute a code c to get a new pass rate, it is backpropagated to the parent nodes, and all thought steps along the path update their corresponding $p_{\text{pass}}^{\text{priv}}$ with the newly observed pass rate.

$$p_{\text{pass}}^{\text{priv}}(s) = \frac{1}{|\mathcal{R}_c|} \sum_{c \in \mathcal{R}_c} p_{\text{pass}}^{\text{priv}}(c), \tag{2}$$

where $\mathcal{R}_c$ is the set of all code that descended from s and its child nodes.

Final Data for Training CodePRM Following the procedure outlined above, we store each data sample as $(x, \{t_1, t_2, ..., t_i\}, c, f, p_{\text{pass}}^{\text{priv}})$. By collecting a large number of such samples from various problems, we create a labeled dataset suitable for training CodePRM. Each partial reasoning process is linked to its corresponding code and execu-

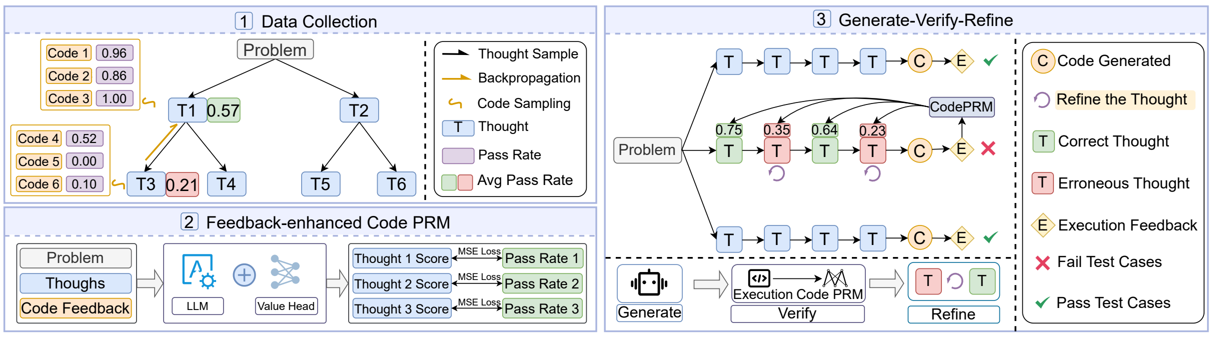


Figure 2: Overview of the data collection, training and inference of CODEPRM. We collect data labeled with average pass rates via tree search to train an execution feedback-enhanced PRM. For evaluation, we integrate CODEPRM into a Generate-Verify-Refine (GVR) pipeline, where it verifies and refines thought steps, improving the quality of generated thoughts and code.

tion feedback, with a label driven by the pass rates of all subsequent code.

3.2 CODEPRM: Feedback-enhanced PRM for Code

We employ supervised fine-tuning to train an LLM to function as a code PRM. Unlike other PRMs that take *{problem description, thought steps}* as input, we enhance the input by incorporating the corresponding code and execution feedback for the thoughts as *{problem description, thought steps, code, execution feedback}*. The LLM, augmented with a value head, then outputs the predicted score for each thought. The training label is the average pass rate, and we train the PRM using Mean Squared Error (MSE) loss.

$$\mathcal{L} = \frac{1}{|\mathcal{D}|}\sum_{j=1}^{|\mathcal{D}|}((\mathrm{PRM}(x, \{t\}_{i=1}^{j}, c, f) - p_{\text{pass},j}^{\text{priv}}))^2, \tag{3}$$

where x is the problem description; $\{t\}_{i=1}^{j}$ is the thoughts; c and f represent the code and execution feedback; $p_{\text{pass},j}^{\text{priv}}$ is the average pass rate of the thought j.

3.3 Generate-Verify-Refine

The key advantage of CODEPRM is its ability to identify errors in the reasoning process through the execution feedback of the code. We propose a Generate-Verify-Refine (GVR) inference pipeline, where CODEPRM serves as a thought process verifier that identifies and refines real-time thought errors during code generation, enabling a dynamic and iterative search process that continuously improves the output.

3.3.1 Generate

First, we employ a generator LLM to explore different thoughts and strategies for solving a code problem. The generator LLM is provided with the *problem description* and instructed to propose W candidate thought steps. Iteratively, different thought chains can be formed and corresponding code is generated. Various tree search algorithms, such as Best-of-N and Monte Carlo Tree Search (MCTS), can be used for exploration. In the Best-of-N approach, the algorithm first generates N thought chains of fixed length, each leading to a final code. For MCTS, every node in the tree maintains its corresponding code c_i, and the algorithm dynamically explores the tree by balancing exploration and exploitation. Details of these two algorithms can be found in Appendix B.

3.3.2 Verify

After the generation phase, we obtain a set of initial codes from the search process. For codes that successfully pass all public test cases, they will not be further processed. However, for code that fails in certain public test cases, we analyze and verify the thought steps. To achieve this, we leverage CODEPRM as a verifier to evaluate the quality of each thought step. The goal is to identify low-quality thoughts that may have led to errors in the code. CODEPRM takes as input the problem description x, the thought chain $T = (t_1, t_2, ..., t_i)$, the corresponding code c, and the execution feedback f, and outputs a quality score q_i for each thought step t_i. Formally, the input can be represented as:

$$(q_1, q_2, ..., q_i) = PRM(x \oplus T \oplus c \oplus f), \tag{4}$$

where $\oplus$ denotes concatenation; $q_i \in [0, 1]$ represents the quality of the i-th thought step.

Using the scores, we explore two verification strategies to determine which thoughts require refinement:

- Threshold-based Verification: A predefined threshold θ is set (e.g. $\theta = 0.5$). Any thought steps with a score $q_i \leq \theta$ is flagged as requiring correction. This approach ensures that all steps below a certain quality standard are refined.
- Locate-Minimum Verification. Only the thought step t_{min} with the lowest score q_{min} is identified for correction. This method focuses on the most critical flaw in the thought trace, optimizing the refinement process by targeting the weakest link.

We compare the locate-minimum approach because our experiments revealed that refining more thoughts does not necessarily improve results when using the LLM. Additionally, the threshold θ in the threshold-based method requires parameter engineering to determine its optimal value. Therefore, we believe that the locate-minimum strategy is a compelling alternative worth exploring and comparing.

The scoring of intermediate thought steps by CodePRM serves not only to verify the correctness of these steps but can also be directly applied in search algorithms as reward to filter results. In MCTS, the PRM's scores act as rewards for nodes, guiding the search process. Similarly, in the Best-of-N algorithm, these scores are used to select the optimal path from multiple candidates. This dual functionality enhances the efficiency and accuracy of both search and selection processes.

3.3.3 Refine

After identifying the thoughts requiring refinement through the "verify" step, the generator is tasked with refining these thoughts to correct previous errors. Formally, the generator is is provided with a prompt P_{refine} that includes the *problem description* x, thought trace $T = (t_1, t_2, ..., t_i)$, code c and execution feedback f.

$$P_{\text{refine}} = x \oplus T \oplus c \oplus f \tag{5}$$

The generator LLM is then instructed to refine a subset of thoughts $T_{\text{refine}} \subseteq T$ where $T_{\text{refine}} = \{t_j | t_j \text{ is flagged for refinement}\}$. The refinement process can be represented as:

$$T_{\text{refined}} = \text{Generator}(P_{\text{refine}}, T_{\text{refine}}) \tag{6}$$

The generator focuses on modifying the specified thoughts T_{refine} to address the errors highlighted by f, ensuring that the revised thought chain T_{refined} leads to improved code. This targeted refinement process ensures that the generator efficiently corrects errors while maintaining the coherence and logical flow of the overall solution.

4 Experiments

In this section, we conduct experiments to illustrate the effectiveness of CodePRM compared with previous methods and investigate the properties of building and using a code PRM.

4.1 Experimental Setup

Datasets Our evaluation encompasses two widely used datasets: APPS (Hendrycks et al., 2021) and Codeforces (MatrixStudio, 2024), more details can be found in Appendix A:

- **APPS** contains three levels of difficulties: introductory, interview, and competition. We evaluate all the methods on the formal 100 problems of each difficulty.
- **Codeforces** provide each problem with a rating score indicating the difficulty. We pick three levels of difficulties 1200, 1500, and 1700, and evaluate each method on the formal 100 problems of each difficulty.

Baselines To illustrate the effectiveness of CodePRM with the Generate-Verify-Refine (GVR) inference pipeline, we compare two kinds of code generation methods. The first kind is feedback-enhanced, which uses the code execution feedback to refine code iteratively: LDB (Zhong et al., 2024), Reflexion (Shinn et al., 2024). The second kind is tree search-enhanced methods: PG-TD (Zhang et al., 2023), ToT (Yao et al., 2024), LATS (Zhou et al., 2023), RAP (Hao et al., 2023) and RethinkMCTS (Li et al., 2024b). We compared various methods on two models: the closed-source GPT-4o-mini (OpenAI, 2024a) and the open-source DeepSeek-Coder-V2-Lite-Instruct (Zhu et al., 2024).

Implementation Details For search-based methods, we configure each node to have 3 child nodes. In Table 1, we implement CodePRM and the GVR pipeline within the MCTS algorithm. We limit the maximum number of rollouts or code generated to 16. For fine-tuning CodePRM, we selected Qwen2.5-Coder-7B-Instruct as the base model, balancing computational cost and effectiveness to develop CodePRM. During the training of Code-

PRM, we utilized 600 problems from APPS, ensuring that none of these training problems appears in the inference testing phase.

Evaluation Metircs We adopt *pass rate* and *pass@1* as the primary metrics to evaluate code correctness, following the methodology outlined in (Zhang et al., 2023). *Pass rate* represents the average proportion of private test cases successfully passed by the generated code across all problems. *pass@1* measures the percentage of problems for which the generated programs pass all private test cases, which is the most widely used metric in the literature of code generation (Chen et al., 2021).

4.2 Code Generation Performance

We present a comparison between CodePRM(GVR-MCTS) and baseline methods, with results shown in Table 1. CodePRM(GVR-MCTS) outperforms other tree search and code generation methods. Among all methods, RethinkMCTS and CodePRM(GVR-MCTS) achieve leading results for having refinement mechanisms. This demonstrates that incorporating refinement in tree search can lead to better search outcomes by guiding the search process along more effective paths. Compared to RethinkMCTS, CodePRM(GVR-MCTS) performs better by using CodePRM as a verifier to identify which thoughts should be refined, demonstrating that a well-trained code PRM can enhance code searches.

4.3 Empirical Analysis

(1) For fine-tuning a code PRM, is it necessary to have code execution feedback as input? Figure 3 presents a comparative analysis of various fine-tuned Qwen2.5-Coder-7B-Instruct models as code PRMs. "Random Reward" means randomly assigned rewards used for BoN candidate selection. "Qwen Reward" means rewards provided by an untuned, vanilla Qwen model via prompting, serving as the filtering criterion for BoN candidate selection. "Normal PRM Reward" means rewards from from PRM trained in a traditional way (without execution feedback, 0/1 labeling, etc), used as the filtering criterion for Best-of-N candidate selection. The experimental results reveal that fine-tuning an LLM as a PRM without the code execution feedback yields performance comparable to random reward, demonstrating that it's not suitable to train a PRM to evaluate thought steps in code independently. The code and execution feedback are essential. Moreover, the poor performance of the original Qwen2.5-Coder-7B-Instruct model as a PRM further substantiates that code reward modeling represents a specialized domain requiring dedicated fine-tuning to achieve competent performance.

(2) For fine-tuning a code PRM, is it suitable to use the pass rate as labels? We conducted a comparative study on fine-tuning CodePRM using two labeling strategies: (1) binary 0/1 labels and (2) pass rate as labels. The resulting PRMs were then applied to Best-of-N inference, with the performance comparison presented in Figure 4. Our findings demonstrate that CodePRM fine-tuned with pass rate labels consistently improves the overall pass rate while maintaining comparable performance on the pass@1 metric compared to binary labels.

This improvement can be attributed to two key factors. First, transitioning from hard labels (binary 0/1) to soft labels (continuous pass rates) provides richer supervisory signals during training. This is similar to using averaged binary labels as soft targets, a technique commonly used in mathematical reasoning tasks. Second, and more importantly, the pass rate label serves as a more effective predictor of "the quality of code that will be generated from a given thought." In contrast, binary labels primarily assess "the correctness of the thought itself." This distinction is particularly crucial in code generation tasks, where the relationship between a thought and its resulting code is not strictly deterministic. For instance, a thought step may produce multiple code snippets, each with a pass rate above 0.9. Under a binary labeling scheme, this thought step would be labeled as 0, despite its potential to generate high-quality code. While such labeling may work well in mathematical reasoning, it is less suitable for code generation. Therefore, pass rates are a more appropriate metric for constructing a code PRM.

The experimental results validate that this approach better aligns with the inherent characteristics of code generation tasks, where the probabilistic nature of thought-to-code translation necessitates a more nuanced evaluation metric than simple binary classification.

(3) For applying a code PRM in GVR as a verifier, does the threshold-based approach outperform the locate-minimum strategy? In this study, we investigate the comparative effectiveness of two verification strategies: the threshold-based approach and the locate-minimum approach.

Model	APPS						Codeforces					
	Pass Rate (%)			Pass@1 (%)			Pass Rate (%)			Pass@1 (%)		
	Intro.	Inter.	Comp.	Intro.	Inter.	Comp.	1200	1500	1700	1200	1500	1700
GPT-4o-mini												
Base	56.56	52.40	35.00	35	29	16	61.53	50.75	40.99	40	26	18
PG-TD	65.87	70.37	43.16	45	46	27	76.78	67.32	56.48	61	44	28
ToT	71.03	67.84	37.17	52	46	23	79.29	68.39	60.10	63	43	35
LATS	69.46	67.65	35.83	50	45	19	77.16	65.12	55.82	67	42	28
RAP	64.24	57.25	37.67	39	32	20	67.13	50.44	44.23	45	26	20
LDB	60.64	60.78	40.33	40	38	23	80.01	62.05	52.43	68	40	30
Reflexion	60.65	56.87	38.00	40	31	18	77.20	66.23	52.39	61	41	26
RethinkMCTS	76.60	74.35	42.50	59	49	28	83.23	67.68	61.73	70	42	40
CodePRM(GVR-MCTS)	**77.20**	**76.33**	**48.17**	**62**	**57**	**29**	**85.29**	**71.28**	**64.46**	**73**	**48**	**41**
Deepseek-Coder-V2-Lite-Instruct												
Base	47.67	45.89	19.33	27	23	6	51.02	34.95	26.95	30	12	11
PG-TD	61.52	57.53	24.50	41	31	7	62.42	50.90	41.31	45	27	16
ToT	63.67	59.49	28.00	40	32	10	66.92	48.57	44.93	53	26	21
LATS	64.70	62.73	28.67	48	38	13	68.57	51.99	49.18	51	32	24
RAP	47.20	35.92	22.33	28	12	9	48.08	35.91	29.17	30	17	12
LDB	56.67	50.85	19.50	37	27	6	60.99	45.90	34.84	42	24	16
Reflexion	53.17	48.72	19.50	32	26	6	51.72	40.95	32.79	37	16	15
RethinkMCTS	68.57	67.95	28.50	50	41	11	70.68	53.83	49.05	53	32	24
CodePRM(GVR-MCTS)	**70.03**	**71.69**	**31.00**	**53**	**46**	**17**	**70.80**	**61.12**	**53.98**	**56**	**40**	**27**

Table 1: Performance comparison between different models and strategies. We report pass rate and pass@1 on both datasets and all difficulties. The maximum number of rollouts for tree search algorithms being 16.

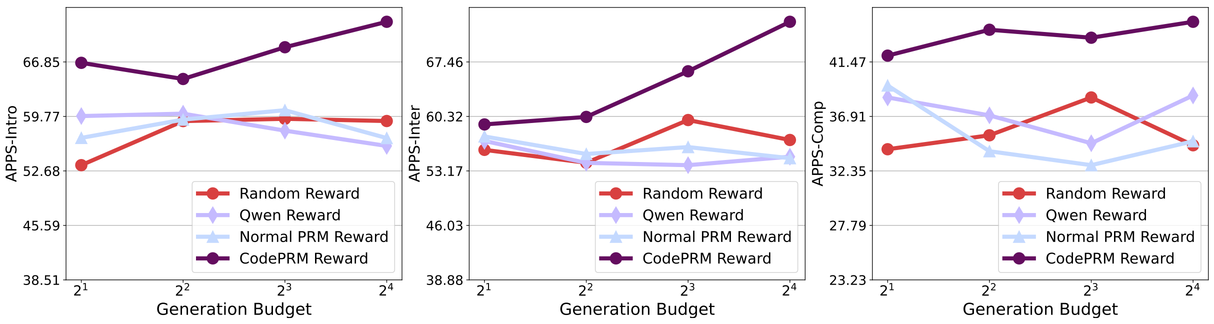
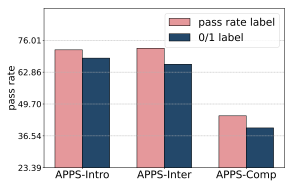
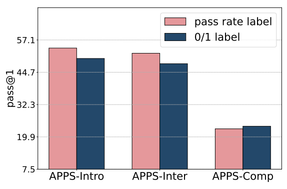


Figure 3: Performance comparison using different fine-tuned Qwen2.5-Coder-7B-Instruct as code PRMs. We assess their effectiveness by using them to provide reward in Best-of-N.



(a) Pass Rate Comparison



(b) Pass@1 Comparison

Figure 4: Performance comparison of fine-tuning CodePRM using different labels. We experiment with two types of labels fine-tuned CodePRM on GVR-MCTS.

As demonstrated in Table 2, for simpler tasks (APPS-Intro/Inter), both methods achieve comparable results, suggesting locate-minimum correction suffices in sparse-error scenarios. While in APPS-Comp, threshold-based method under low-precision verifiers (GPT-4o-mini) may cause overcorrection, and on CodePRM, threshold-based verification shows superior performance. It indicates that with more precise evaluation scores, the threshold-based method could show its advantage. This reveals an adaptive principle: verification strategies should dynamically align with task complexity and verifier reliability.

Dataset	Locate-Min (GPT-4o-mini)	Threshold (GPT-4o-mini)	Locate-Min (CodePRM)	Threshold (CodePRM)
APPS-Intro.	74.05	**75.53**	76.41	**77.20**
APPS-Inter.	76.41	**77.63**	76.31	**76.33**
890 APPS-Comp.	**43.50**	39.83	42.50	**48.17**

Table 2: Comparison between locate-minimum and threshold-based verification approaches. We experiment on GPT-4o-mini and CodePRM as the verifier in GVR-MCTS.

(4) For applying CodePRM as verifier, does refining more thoughts lead to better performance? The threshold-based approach demonstrates strong efficacy when employing code PRM as a verifier. This prompts the question: does this effectiveness stem from the threshold's capacity to modify more thoughts? In this study, we delve into this inquiry. Figure 5 illustrates the variation in performance as the threshold changes. It becomes evident that a higher threshold—implying more thoughts are refined—does not necessarily lead to superior performance. This phenomenon may be attributed to two reasons. Firstly, from the perspective of refinement, excessive refinement may transform correct thoughts into erroneous ones. Secondly, from the verifier's perspective, the accuracy of discrimination may vary across different scoring intervals. For instance, distinguishing between thoughts scored above 0.9 may be challenging in terms of determining which is superior or inferior, whereas thoughts scored above or below 0.5 may be more readily discernible in quality.

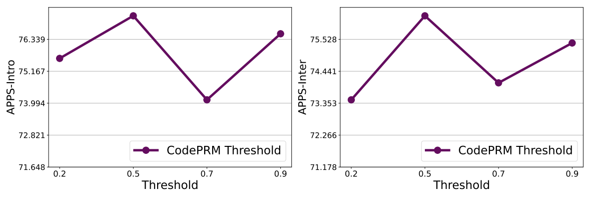


Figure 5: Performance changes as the threshold varies in GVR-MCTS using CodePRM as the verifier. It's not the case that the more thoughts to be refined (or the higher the threshold), the better the performance.

(5) For general LLMs, could they be directly used as code PRMs with code and execution-feedback enhancement? Here, we conduct experiments on CodePRM, GPT-4o-mini, and the original Qwen2.5-Coder-7B-Instruct to investigate if general LLMs could be directly used as code PRMs with the code and execution-feedback enhancement. The results are shown in Figure 6. We can see that the code and execution feedback could bring obvious performance improvement in general LLMs. Despite that, CodePRM outperforms them as a code PRM, demonstrating that providing process rewards is a highly specialized task. Fine-tuning can effectively enhance its performance, surpassing that of general LLMs. Furthermore, for general LLMs, the evaluation performance is poorer when there is no code execution feedback in the input. This indicates that, in a coding environment, the assessment of thought quality is closely tied to the generated code.

(6) Comparing with open-sourced PRMs, does CodePRM perform better in code generation? Here, we compare the performance of CodePRM with that of open-source PRMs. These

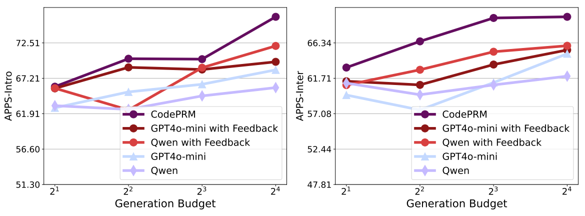


Figure 6: Performance comparison between various input types for LLMs functioning as code PRMs in Best-of-N. It can be seen that incorporating execution feedback as an input significantly enhances the model's performance in process evaluation.

PRMs are utilized to provide rewards in a Best-of-N algorithm based on GPT-4o-mini, and the results are presented in Table 3. The Skywork-PRM-1.5B (o1 Team, 2024a) and Skywork-PRM-7B (o1 Team, 2024b) models are general-purpose PRMs, applicable to both code and mathematics. Since most open-source PRMs are specialized in mathematics, we also included Math-Shepherd-Mistral-7B (Wang et al., 2024c) for comparison. The results indicate that CodePRM significantly outperforms these open-source PRMs in code generation tasks. The primary reason is that while open-source PRMs can directly assess the correctness of intermediate reasoning steps, constructing a code PRM requires the inclusion of both code and feedback in the input, which is crucial for evaluating the thought steps for code.

PRM	APPS					
	Pass Rate (%)			Pass@1 (%)		
	Intro.	Inter.	Comp.	Intro.	Inter.	Comp.
MathShepherd-Mistral-7B	61.66	58.90	36.33	37	37	21
Skywork-PRM-1.5B	59.54	60.17	39.17	38	37	21
Skywork-PRM-7B	61.81	59.31	35.17	38	35	17
CodePRM-7B	**72.07**	**72.72**	**44.83**	**54**	**52**	**23**

Table 3: Performance comparison with open-source PRMs. We report pass rate and pass@1 on APPS dataset and the maximum number of rollouts for Best-of-N algorithm being 16.

5 Conclusion

In this paper, we introduced CodePRM, a novel framework that leverages execution feedback to build a process reward model for code generation. Our approach collects thought steps enriched with code pass rates, code snippets, and execution feedback, and trains a feedback-enhanced code PRM to score individual thought steps. At inference time, CodePRM serves as a process verifier within a Generate-Verify-Refine (GVR) pipeline, enabling dynamic error correction during code search. Experimental results demonstrate that CodePRM significantly outperforms strong baselines. Further analysis identifies key factors for building effective code PRMs, such as the importance of execution feedback and the granularity of thought steps.

Looking ahead, CodePRM opens new avenues for enhancing the reasoning capabilities of LLMs in complex programming tasks. Future work could explore extending this framework to other domains requiring high-level reasoning, as well as investigating more sophisticated methods for integrating execution feedback into the training and inference processes.

Ackowledgement

The Shanghai Jiao Tong University team is partially supported by National Key R&D Program of China (2022ZD0114804), Shanghai Municipal Science and Technology Major Project (2021SHZDZX0102) and National Natural Science Foundation of China (62322603, 62177033).

Limitations

Generalization to Other Reasoning Tasks Our primary contribution lies in developing PRM for code that leverage execution feedback. While this approach is effective for code generation tasks, it may not generalize well to other reasoning tasks, such as mathematical reasoning, where traditional format PRMs have shown strong performance (Ma et al., 2023; Li and Li, 2024). However, for reasoning tasks that involve detailed feedback mechanisms similar to code generation, our method could potentially be applicable.

Limited Exploration of Fine-Tuning Other LLMs Our experiments, including both standard PRM and feedback-enhanced PRM, were conducted using the fine-tuned Qwen2.5-Coder-7B-Instruct model. While this provides a solid foundation, future work could explore the application of our methods to other large language models to assess their broader applicability and performance.

PRMs in Reinforcement Learning Training Although PRMs have the potential to offer dense, step-level rewards in Reinforcement Learning (RL) training scenarios (Dai et al., 2024; Zhang et al., 2024), our current work focuses primarily on the application of PRMs in inference algorithms. The integration of PRMs into RL training frameworks remains an open area for future research.

Potential Errors in Refinement Step Our current framework emphasizes the development of a verifier that identifies thought steps requiring refinement, delegating the actual refinement process to the generator LLM. This approach, while effective, carries the risk of incorrect refinements. Developing a robust thought refiner to mitigate such errors represents a promising direction for future research.

Ethics Statement

In this work, we employ LLMs as both thought and code generators, and we fine-tune certain LLMs to serve as process reward models (PRMs). All the dataset we use to fine-tune LLMs are publicly available and are for research purposes only. The LLMs utilized in our study include open-source models such as Deepseek-Coder-V2-Lite-Instruct and Qwen2.5-Coder-7B-Instruct, as well as the closed-source model GPT-4o-mini. Ethical considerations related to these models, including their training data and deployment, are addressed by their respective creators. The fine-tuned PRMs in our work are designed solely to output evaluation scores for thought steps and do not generate free-form text. However, we acknowledge that LLMs, including those used in our study, may occasionally produce improper or harmful content. Such outputs are unintended and do not reflect the views or intentions of the authors.

References

Eshwar Ram Arunachaleswaran, Natalie Collina, and Jon Schneider. 2024. Pareto-optimal algorithms for learning in games. In *Proceedings of the 25th ACM Conference on Economics and Computation*, pages 490–510.

Jingchang Chen, Hongxuan Tang, Zheng Chu, Qianglong Chen, Zekun Wang, Ming Liu, and Bing Qin. 2024. Divide-and-conquer meets consensus: Unleashing the power of functions in code generation. *arXiv preprint arXiv:2405.20092*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128*.

Ning Dai, Zheng Wu, Renjie Zheng, Ziyun Wei, Wenlei Shi, Xing Jin, Guanlin Liu, Chen Dun, Liang Huang, and Lin Yan. 2024. Process supervision-guided policy optimization for code generation. *arXiv preprint arXiv:2410.17621*.

Lingyue Fu, Huacan Chai, Shuang Luo, Kounianhua Du, Weiming Zhang, Longteng Fan, Jiayi Lei, Renting Rui, Jianghao Lin, Yuchen Fang, et al. 2023. Codeapex: A bilingual programming evaluation benchmark for large language models. *arXiv preprint arXiv:2309.01940*.

Shibo Hao, Yi Gu, Haodi Ma, Joshua Jiahua Hong, Zhen Wang, Daisy Zhe Wang, and Zhiting Hu. 2023. Reasoning with language model is planning with world model. *arXiv preprint arXiv:2305.14992*.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, et al. 2021. Measuring coding challenge competence with apps. *arXiv preprint arXiv:2105.09938*.

Nan Jiang, Xiaopeng Li, Shiqi Wang, Qiang Zhou, Soneya Binta Hossain, Baishakhi Ray, Varun Kumar, Xiaofei Ma, and Anoop Deoras. 2024. Training llms to better self-debug and explain code. *arXiv preprint arXiv:2405.18649*.

Cuong Chi Le, Hoang-Chau Truong-Vinh, Huy Nhat Phan, Dung Duy Le, Tien N Nguyen, and Nghi DQ Bui. 2024. Visualcoder: Guiding large language models in code execution with fine-grained multimodal chain-of-thought reasoning. *arXiv preprint arXiv:2410.23402*.

Hung Le, Hailin Chen, Amrita Saha, Akash Gokul, Doyen Sahoo, and Shafiq Joty. 2023. Codechain: Towards modular code generation through chain of self-revisions with representative sub-modules. *arXiv preprint arXiv:2310.08992*.

Jierui Li, Hung Le, Yinbo Zhou, Caiming Xiong, Silvio Savarese, and Doyen Sahoo. 2024a. Codetree: Agent-guided tree search for code generation with large language models. *arXiv preprint arXiv:2411.04329*.

Qingyao Li, Wei Xia, Kounianhua Du, Xinyi Dai, Ruiming Tang, Yasheng Wang, Yong Yu, and Weinan Zhang. 2024b. Rethinkmcts: Refining erroneous thoughts in monte carlo tree search for code generation. *arXiv preprint arXiv:2409.09584*.

Wendi Li and Yixuan Li. 2024. Process reward model with q-value rankings. *arXiv preprint arXiv:2410.11287*.

Zhenwen Liang, Ye Liu, Tong Niu, Xiangliang Zhang, Yingbo Zhou, and Semih Yavuz. 2024. Improving llm reasoning through scaling inference computation with collaborative verification. *arXiv preprint arXiv:2410.05318*.

Jianqiao Lu, Zhiyang Dou, WANG Hongru, Zeyu Cao, Jianbo Dai, Yunlong Feng, and Zhijiang Guo. 2024. Autopsv: Automated process-supervised verifier. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*.

Liangchen Luo, Yinxiao Liu, Rosanne Liu, Samrat Phatale, Harsh Lara, Yunxuan Li, Lei Shu, Yun Zhu, Lei Meng, Jiao Sun, et al. 2024. Improve mathematical reasoning in language models by automated process supervision. *arXiv preprint arXiv:2406.06592*.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. Wizardcoder: Empowering code large language models with evol-instruct. *arXiv preprint arXiv:2306.08568*.

Qianli Ma, Haotian Zhou, Tingkai Liu, Jianbo Yuan, Pengfei Liu, Yang You, and Hongxia Yang. 2023. Let's reward step by step: Step-level reward model as the navigators for reasoning. *arXiv preprint arXiv:2310.10080*.

MatrixStudio. 2024. Codeforces-python-submissions.

Ansong Ni, Miltiadis Allamanis, Arman Cohan, Yinlin Deng, Kensen Shi, Charles Sutton, and Pengcheng Yin. 2024. Next: Teaching large language models to reason about code execution. *arXiv preprint arXiv:2404.14662*.

Skywork o1 Team. 2024a. Skywork-prm 1.5b. `https://huggingface.co/Skywork/Skywork-o1-Open-PRM-Qwen-2.5-1.5B`.

Skywork o1 Team. 2024b. Skywork-prm 7b. `https://huggingface.co/Skywork/Skywork-o1-Open-PRM-Qwen-2.5-7B`.

OpenAI. 2024a. Gpt-4o-mini. `https://openai.com/index/gpt-4o-mini-advancing-cost-efficient-intelligence/`. Accessed: 2024-07-18.

OpenAI. 2024b. Openai o1 system card. Accessed: 2025-02-07.

Amrith Setlur, Chirag Nagpal, Adam Fisch, Xinyang Geng, Jacob Eisenstein, Rishabh Agarwal, Alekh Agarwal, Jonathan Berant, and Aviral Kumar. 2024. Rewarding progress: Scaling automated process verifiers for llm reasoning. *arXiv preprint arXiv:2410.08146*.

Bo Shen, Jiaxin Zhang, Taihong Chen, Daoguang Zan, Bing Geng, An Fu, Muhan Zeng, Ailun Yu, Jichuan Ji, Jingyang Zhao, et al. 2023. Pangu-coder2: Boosting large language models for code with ranking feedback. *arXiv preprint arXiv:2307.14936*.

Yuling Shi, Songsong Wang, Chengcheng Wan, and Xiaodong Gu. 2024. From code to correctness: Closing the last mile of code generation with hierarchical debugging. *arXiv preprint arXiv:2410.01215*.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2024. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36.

David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. 2017. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*.

Codeforces Team. 2024. Codeforces. https://codeforces.com/.

Evan Wang, Federico Cassano, Catherine Wu, Yunfeng Bai, Will Song, Vaskar Nath, Ziwen Han, Sean Hendryx, Summer Yue, and Hugh Zhang. 2024a. Planning in natural language improves llm search for code generation. *arXiv preprint arXiv:2409.03733*.

Jun Wang, Meng Fang, Ziyu Wan, Muning Wen, Jiachen Zhu, Anjie Liu, Ziqin Gong, Yan Song, Lei Chen, Lionel M Ni, et al. 2024b. Openr: An open source framework for advanced reasoning with large language models. *arXiv preprint arXiv:2410.09671*.

Peiyi Wang, Lei Li, Zhihong Shao, Runxin Xu, Damai Dai, Yifei Li, Deli Chen, Yu Wu, and Zhifang Sui. 2024c. Math-shepherd: Verify and reinforce llms step-by-step without human annotations. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 9426–9439.

Zihan Wang, Yunxuan Li, Yuexin Wu, Liangchen Luo, Le Hou, Hongkun Yu, and Jingbo Shang. 2024d. Multi-step problem solving through a verifier: An empirical analysis on model-induced process supervision. *arXiv preprint arXiv:2402.02658*.

Weimin Xiong, Yifan Song, Xiutian Zhao, Wenhao Wu, Xun Wang, Ke Wang, Cheng Li, Wei Peng, and Sujian Li. 2024. Watch every step! llm agent learning via iterative step-level process refinement. *arXiv preprint arXiv:2406.11176*.

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.

Zhuohao Yu, Weizheng Gu, Yidong Wang, Zhengran Zeng, Jindong Wang, Wei Ye, and Shikun Zhang. 2024. Outcome-refining process supervision for code generation. *arXiv preprint arXiv:2412.15118*.

Hanning Zhang, Pengcheng Wang, Shizhe Diao, Yong Lin, Rui Pan, Hanze Dong, Dylan Zhang, Pavlo Molchanov, and Tong Zhang. 2024. Entropy-regularized process reward model. *arXiv preprint arXiv:2412.11006*.

Shun Zhang, Zhenfang Chen, Yikang Shen, Mingyu Ding, Joshua B Tenenbaum, and Chuang Gan. 2023. Planning with large language models for code generation. *arXiv preprint arXiv:2303.05510*.

Zhenru Zhang, Chujie Zheng, Yangzhen Wu, Beichen Zhang, Runji Lin, Bowen Yu, Dayiheng Liu, Jingren Zhou, and Junyang Lin. 2025. The lessons of developing process reward models in mathematical reasoning. *arXiv preprint arXiv:2501.07301*.

Li Zhong, Zilong Wang, and Jingbo Shang. 2024. Ldb: A large language model debugger via verifying runtime execution step-by-step. *arXiv preprint arXiv:2402.16906*.

Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. 2023. Language agent tree search unifies reasoning acting and planning in language models. *arXiv preprint arXiv:2310.04406*.

Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y Wu, Yukun Li, Huazuo Gao, Shirong Ma, et al. 2024. Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence. *arXiv preprint arXiv:2406.11931*.

Appendix

A Experiment Details

Dataset Details Here we present some details about the details of two datasets we utilize: APPS and Codeforces. The APPS dataset comprises programming problems divided into three difficulty tiers: introductory, interview, and competition. It includes 5000 problems in a training set and an additional 5000 in the testing set. On the other hand, the Codeforces dataset consists of problems sourced from the Codeforces online programming contests (Team, 2024), with difficulty levels classified by "ratings". For evaluation purposes, we selected problems with ratings of 1200, 1500, and 1700. Since the both datasets does not distinguish between public and private test cases, we split each problem's test cases evenly into two sets, following the approach of Zhang et al. (2023). The first set is used as the public test cases that can be seen during the algorithm running, and the second set is used as the private test cases for evaluating the generated codes.

Additional Implementation Details In Table 1, the CODEPRM is used to verify intermediate reasoning steps and provide additional rewards beyond the code's pass rate on public test cases. For using CODEPRM to score thoughts, we separate the thought steps with a step tag "\n\n\n\n\n". In

MCTS, each node is corresponded with a generated code. In Best-of-N, we set each thought chain to stop at 4 steps, and corresponding code are generated.

Baseline Details Here, we present more details of the implementation of the baselines:

(1) Code Generation Algorithms:

- **LDB** (Zhong et al., 2024): A debugging framework that divides the initial code into blocks, analyzes each block, and resolves issues by monitoring changes in block-level variable values. It iteratively optimizes the code by following this process.

- **Reflexion** (Shinn et al., 2024):Iteratively refine the initial code by utilizing historical error data and incorporating insights gained from previous errors.

(2) Tree Search-enhanced Methods:

- **PG-TD** (Zhang et al., 2023): A token-level MCTS search method that uses the code's pass rate as a scalar reward.

- **ToT** (Yao et al., 2024): We apply the Tree-of-Thoughts (ToT) approach to code generation in a manner similar to its application in creative writing. The search process is structured into two distinct phases: thought generation and code generation, with the tree being explored using a breadth-first search (BFS) strategy.

- **LATS** (Zhou et al., 2023): A framework that integrates MCTS with reflection, summarizing past errors and storing them as memory within nodes to assist with future iterations.

- **RAP** (Hao et al., 2023): Leveraging an LLM as the world model to simulate and evaluate search results.

- **RethinkMCTS** (Li et al., 2024b): Introducing a refinement mechanism in thought-level MCTS for code, where the most recent thought in the tree is refined when the generated code fails some public test cases.

B Tree Search Methods

In our experiment, we employ two tree search algorithms: Best-of-N and Monte Carlo Tree search. Here we introduce the details of these two methods.

Best-of-N Best-of-N is a sampling strategy that generates N candidate thought trajectories from a generator LLM. Then each thought trajectory would further derive a corresponding code. The final output is selected by maximizing the reward from the process reward model (PRM). This method trades computational cost (linear in N) for improved output quality through parallel candidate evaluation.

Monte Carlo Tree Search (MCTS) MCTS balances exploration and exploitation through four iterative phases:

- **Selection:** We employ P-UCB (Silver et al., 2017), an enhanced version of the UCB algorithm, to compute the overall score for each node:

$$\text{P-UCB}(s,a) = Q(s,a) + \beta(s)\cdot p(a \mid s) \cdot \frac{\sqrt{\log(N(s))}}{1+N(s')}, \tag{7}$$

  where s' is the state reached by taking action a in s; $N(s)$ is the visited times of the node; $p(a \mid s)$ is the probability that thought a is the next thought given the problem description and previous thoughts s, which is proposed by the generator LLM. β is the weight for exploration, which depends on the number of visit of s, defined as

$$\beta(s) = \log\left(\frac{N(s)+c_{\text{base}}+1}{c_{\text{base}}}\right) + c, \tag{8}$$

  where c_{base} is a hyperparameter; c is the exploration weight.

- **Expansion:** Create child nodes when reaching expandable leaf nodes.

- **Simulation:** Generate code base on the previous thought steps from the root node to current node.

- **Backpropagation:** It updates the Q-values and visited times of all nodes along the path from the current node to the root node using the rewards obtained from the evaluation.

MCTS dynamically allocates computation to promising search paths, achieving Pareto-optimal (Arunachaleswaran et al., 2024) performance under constrained budgets.

C Prompts

Here, we outline the key prompts utilized in our generate-verify-refine inference pipeline.

Table 4 displays the prompt employed for generating intermediate thoughts. Table 5 provides the prompt used to generate code based on the thought trace. Finally, Table 6 illustrates the prompt designed for the generator LLM to refine incorrect thoughts flagged by the verifier.

Prompt for Generating Thoughts

{problem statement}

{thoughts}

Above is a problem to be solved by Python program.

* I need you to analyze and provide new thoughts that can lead to the correct solution code.

* If there are previous thoughts provided, please follow them and offer more detailed and further insights, as a detailed thinking or enhancement for previous ones.

* I need you to output {width} possible thoughts. Remember each only contain one possible distinct reasoning but all following previous thoughts if there are.

* Please wrap your response into a JSON object that contains keys 'Thought-i' with i as the number of your thought, and key 'Reasonableness' with the Reasonableness of each thought, which should between 0 1 and the sum should be 1.

* The JSON should be a **list of dicts**, the dicts are split with comma ','.

Example Answers:
```
[
{"Thought-1":" We could use the print function to finish the task in one line: print(2 + 3)", "Reasonableness": 0.7},
{"Thought-2":" We should calculate the problem by setting a=2+3, and then print(a)", "Reasonableness": 0.29},
{"Thought-3":" The problem can't be solved by Python.", "Reasonableness": 0.01}
]
```

Table 4: Prompt for generating thoughts in search methods.

Prompt for Generating the Code

Complete the Python program to solve the problem. Remember to contain the complete program including all the imports and function header in your response.

Also some thoughts are included that you can refer to and build upon when writing the code.

Answer with the code ONLY. No other explanation or words attached!

{problem statement}

{thoughts}

Table 5: Prompt for generating the code following the thoughts in search methods.

Prompt for Refining the Thoughts

{problem statement}
{thoughts}
{code}
{execution feedback}

Above is the combination of problem, thoughts and code.
Each thought is bounded with an id number at the beginning of the thought.
* Revise and enhance the {refine_id}-Thought above in the thoughts by providing a improved new thought to replace it.
* Remember that you only need to provide the new thought in one or two sentences, not the code.

Table 6: Prompt for generating the code following the thoughts in search methods.