RethinkMCTS: Refining Erroneous Thoughts in Monte Carlo Tree Search for Code Generation

Anonymous ACL submission

Abstract

Tree search methods have demonstrated impressive performance in code generation. Previous methods combine tree search with reflection that summarizes past mistakes to achieve iterative improvement. However, these methods face significant challenges. First, they search directly within the code language space, neglecting the underlying reasoning process critical for effective code generation. Second, reflection-based approaches merely accumulate historical errors in memory without providing correct reasoning pathways, making it difficult for subsequent search iterations to identify optimal solutions, resulting in decreased search quality. In this work, we propose RETHINKMCTS, a framework that systematically explores and refines the reasoning process for code generation. Specifically, we employ MCTS to search for thoughts before code generation and integrate MCTS with a refinement mechanism called *rethink*, which incorporates fine-grained code execution feedback to refine erroneous thoughts during the search. It ensures the search path aligns with better reasoning, improving overall search quality. Through extensive experiments, we demonstrate that RETHINKMCTS outperforms previous search-based and feedback-enhanced code generation baselines[1].

1 Introduction

With the impressive capabilities of large language models (LLMs), research has increasingly focused on enhancing their code generation abilities (Luo et al., 2023; Zheng et al., 2023; Gong et al., 2024). Code generation is a reasoning task that requires multiple attempts and iterative corrections to achieve accurate results (Zhou et al., 2025; Bi et al., 2024), hence search algorithms demonstrate particular promise in this domain, achieving state-of-the-art performance (DeLorenzo et al., 2024; Zhang et al., 2023; Kulal et al., 2019; Zhou et al., 2023). Unlike other reasoning tasks, code environments provide rich execution feedback that can be leveraged to improve results. Previous approaches, such as LATS (Zhou et al., 2023), have effectively combined search with the reflection mechanism, enabling search trees to summarize past errors based on feedback and store them in memory to enhance subsequent search performance.

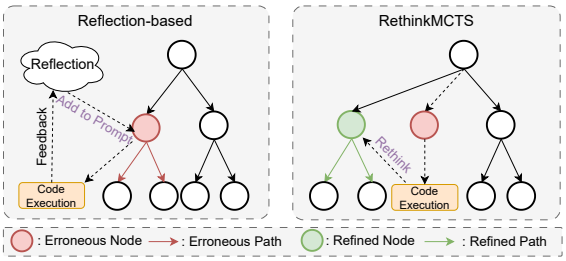

Figure 1: Comparison between reflection-based methods and RETHINKMCTS. Reflection-based methods would maintain the error in the path, while RETHINKMCTS would refine erroneous thoughts and continue along a better path.

Despite demonstrating promising results, previous methods still face two key challenges: 1) Insufficient reasoning exploration. Studies, such as chain-of-thought (Wei et al., 2022) and tree of thoughts (Yao et al., 2024), show that explicitly modeling the reasoning process leads to better results. Tang et al. (2023) further highlighted that LLMs are better equipped for semantic reasoning than symbolic reasoning. However, for code generation, a high-reasoning-demand task (Cook et al., 2018), current work has yet to explore the thoughts (reasoning) behind the generated code. 2) Ineffective error correction. Reflection-based approaches merely accumulate historical errors in memory without providing correct reasoning pathways (Zhou et al., 2023; Shinn et al., 2024), making it difficult for subsequent search iterations to identify optimal solutions, resulting in diminished

[1]Resources are available at https://anonymous.4open.science/r/RethinkMCTS-D748/.

search quality.

This paper presents a novel perspective on the problem by introducing a direct thought revision approach. Previous work (Wang et al., 2024b) has established that correct reasoning processes lead to correct code, and we leverage this insight to achieve accurate code generation through continuous refinement of the underlying thought processes. As illustrated in Figure 1, traditional *reflection* mechanisms merely append historical errors without actively refining the reasoning trajectory, requiring subsequent search algorithms to process increasingly lengthy memory traces. Our approach directly refines erroneous thoughts, enabling the natural emergence of correct reasoning pathways. This targeted refinement strategy significantly improves efficiency by addressing the root causes of errors rather than accumulating extensive error histories.

In light of this, we develop RETHINKMCTS, a thought-search framework for code generation that simultaneously searches and refines reasoning based on code execution feedback. Specifically, RETHINKMCTS begins by employing the MCTS algorithm to explore reasoning paths before generating code and then generates the code based on these reasoning thoughts. After executing the code, we perform a block-level analysis on the code and construct the verbal feedback. Following this, we introduce a refinement mechanism called *rethink*, which makes the LLM refine erroneous thoughts based on the feedback. As shown in Figure 1, this enables the search algorithm to continue exploring along corrected paths, ultimately enhancing the search tree's quality. To further guide action evaluation in the MCTS search process, we propose a dual evaluation approach to ensure effective code selection, particularly when public test cases alone are insufficient. Extensive experiments not only demonstrate the effectiveness of RETHINKMCTS, but also reveal the critical factors enabling successful tree search in code generation. Our main contributions can be summarized as follows:

- **Reasoning-to-Code Search Framework for Code Generation**: Our framework employs a multi-step thinking process combined with code generation using Monte Carlo Tree Search (MCTS) to explicitly explore various strategies for code generation. A combination of verbal and scalar feedback guides the MCTS tree generation. To the best of our knowledge, we are the first to search and refine the thought process behind code to enhance LLMs on code generation.

- **Refining Erroneous Thoughts in MCTS**: We introduce the *rethink* mechanism into MCTS to refine erroneous thoughts using detailed verbal feedback from code execution, allowing the search to follow higher-quality traces. Different from reflection-based methods that summarize past errors without changing current erroneous reasoning, our approach directly refines flawed thoughts, ensuring the search proceeds along more optimal trajectories.

- **Introducing Detailed Feedback and Dual Evaluation for Refinement**: Block-level analysis is introduced as the detailed feedback of code execution, guiding the refinement of faulty thought. Additionally, a dual evaluation method—using both public test cases and LLM self-evaluations—is used to ensure effective code selection, particularly when public test cases alone cannot fully assess the code's correctness.

2 Related Work

LLMs for Code Generation Large language models (LLMs) have been widely applied and developed in the field of code (Nam et al., 2024; Huang et al., 2023a; Li et al., 2024; He et al., 2024). Research on LLMs for code generation falls into two paradigms: (1) *Code-specialized fine-tuning* that enhances syntax understanding through targeted training (Luo et al., 2023; Li et al., 2023; Fried et al., 2022; Roziere et al., 2023). (2) *LLM-as-agent* frameworks where models orchestrate code generation (Ishibashi and Nishimura, 2024; Zhang et al., 2024a; Jin et al., 2024). LDB proposed by Zhong et al. (2024) takes the LLM as a debugger and utilizes block-level decomposition to locate bugs. PG-TD proposed by Zhang et al. (2023) utilizes Monte Carlo Tree Search (Browne et al., 2012) methods combined with the probabilistic output of LLMs to achieve token-level search for code generation. While effective, these approaches neglect explicit modeling of the semantic reasoning essential for complex coding tasks—a gap our work addresses.

Tree Search-enhanced LLMs Tree search methods can improve the reasoning performance of LLMs by exploring various possible paths (Wang et al., 2024a; Meng et al., 2024; Yuan et al., 2024). By designing different action spaces, LLMs can

explore at different levels (Zhang et al., 2023; Hu et al., 2024; Hao et al., 2023). At the implementation level, LATS (Zhou et al., 2023) conducts code-space search while maintaining error logs as reflective memory for subsequent iterations. TS-LLM (Feng et al., 2023) introduces a training-based approach with learned value functions to direct decoding trajectories. While these methods successfully enhance the task-solving abilities of LLMs, they may not fully harness the potential of tree search in code generation tasks. This is largely because many of these approaches focus on token- or code-level searches, overlooking the deeper reasoning process that is critical for tasks like code generation, which require intricate reasoning. Additionally, the detailed execution feedback provided by the code environment has great potential to guide the search process, but these methods fall short of effectively integrating this feedback into the search. In this paper, we focus on leveraging detailed feedback from the code execution environment to guide and refine the thought process, thereby improving the overall quality of exploration.

3 Preliminaries

3.1 Problem Formulation

We focus on competition-level code generation, following the setup established by Zhang et al. (2023). For a given LLM, the input consists of a problem statement P and a set of public test cases T_{pub}, each defined by an input-output pair. The goal is to develop an inference framework that enables the code generation model M to produce the correct code $C \sim M(P, T_{\text{pub}})$ solving the given problem. To rigorously evaluate performance, we maintain hidden private test cases T_{priv} that remain inaccessible during code generation. The primary evaluation metric is the model's ability to pass these private test cases.

3.2 Block-level Code Analysis

Executing buggy code in an executor can only provide standard error information. If the code runs without crashing but produces incorrect outputs, there is often little to no error feedback available. However, since code is quite structured (Chevalier et al., 2007), it is possible to extract detailed execution feedback through a more organized analysis. We follow previous work by Zhong et al. (2024) to get a block-level code analysis.

In static code analysis, the code could be divided into basic blocks (Larus, 1999). A basic block is defined as a linear sequence of code containing a single entry point and a single exit point (Flow, 1994; Alfred et al., 2007). We first acquire the control-flow graph (CFG) of the code, and then a public test case is fed into this graph to produce an execution trace of the test, $[B_1, B_2, ..., B_n]$, where each node within the CFG corresponds to a basic block. We execute these blocks one by one and track all variable state changes in the trace. These blocks and variables are collected and then provided to the LLM to perform a block-level analysis, assessing whether each block is correct or faulty. We show an example of the analysis process in the Appendix C.6.

4 RethinkMCTS

Overview RethinkMCTS is motivated by the need to search and refine the thought process during code generation using feedback from the coding environment, ultimately guiding the LLM toward correct solutions. To accomplish this, we leverage an LLM to generate both thoughts and code, iteratively refining the reasoning based on execution feedback. We employ Monte Carlo Tree Search (MCTS) as our search algorithm to optimally balance exploration and exploitation. Crucially, we introduce a novel *rethink* mechanism that utilizes detailed code execution feedback to identify and refine erroneous thoughts. This approach enables the search to follow improved reasoning paths, thereby enhancing overall search quality. The framework is shown in Figure 2, and we provide the pseudo-code in Algorithm 1 in the Appendix D. Our design has the following key features:

- **Tree Search for Thought Process**: We employ tree search to explore the thought process of writing code. After multiple reasoning steps, code is generated based on the accumulated thoughts.
- **Rethink Mechanism**: We introduce a *rethink* mechanism that leverages feedback from the code execution to refine and improve the quality of the reasoning process.
- **Block-Level Analysis Feedback**: We use block-level analysis of the code as the fine-grained feedback from code execution.
- **Dual Evaluation**: In our evaluation phase, we propose a dual evaluation approach, wherein both public test cases and LLM evaluation are used to

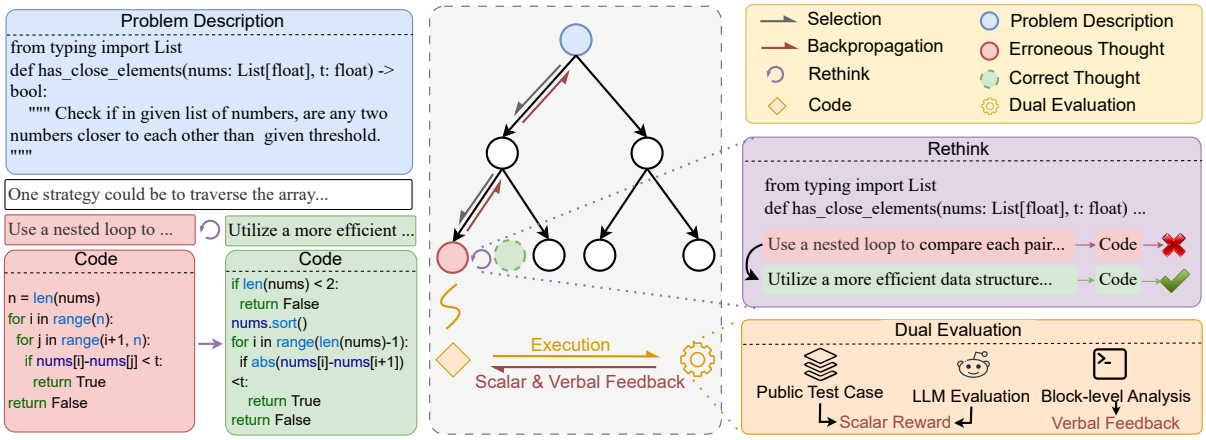

Figure 2: Overview of RETHINKMCTS. We use MCTS to explore different thoughts before generating code. We obtain block-level analysis as verbal feedback through a code executor and use the verbal feedback from failed test cases to refine the thoughts, thereby improving the overall quality of the search tree.

assess the generated code, ultimately helping to identify high-quality solutions.

These key features are integrated into operations in RETHINKMCTS, *selection, expansion, evaluation, verbal feedback, backpropagation, and rethink*.

Selection In MCTS, the selection step balances exploration and exploitation by iteratively choosing the actions that are most promising for further expansion. This process continues until a leaf node is reached. Each node is selected based on a score derived from the number of visits $N(s)$ and the stored value of the state-action pair $Q(s, a)$, where the state s is the problem description and prior thoughts, and action a represents the new thought associated with the node. Every node's retained value $Q(s, a)$ is the maximum reward obtained by starting in s and taking action a. For scoring, we employ P-UCB (Silver et al., 2017), an enhanced version of the UCB algorithm, to compute the overall score for each node:

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

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 LLM agent. β is the weight for exploration, which depends on the number of visit of s, defined as

$$\beta(s) = \log\left(\frac{N(s) + c_{\text{base}} + 1}{c_{\text{base}}}\right) + c, \quad (2)$$

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

At each state or node, the selection process chooses the action with the highest P-UCB value, and repeats this process until a leaf node is reached.

Expansion After selecting a leaf node, the expansion step generates its child nodes to explore different possible actions. We define the search action space as potential thoughts or strategies for writing the code. To make use of the feedback obtained from code execution, we handle the expansion in two scenarios:

- If the current leaf node evaluation has failed public test cases, the expansion step incorporates the verbal feedback f from these failed test cases into the prompt. The LLM then proposes multiple subsequent thoughts z and assigns each thought a reasonableness score e, as represented by $p(a|s)$ in Eq. (1). The output is based on prior thoughts and the current verbal feedback, i.e., $[(z^1, e^1), \ldots, (z^k, e^k)] \sim p((z, e)^{(1\cdots k)}|s, f)$.
- If the current leaf node evaluation passes all public test cases, the expansion step directs the LLM to propose subsequent thoughts without additional feedback, i.e., $[(z^1, e^1), \ldots, (z^k, e^k)] \sim p((z, e)^{(1\cdots k)}|s)$.

After multiple rounds of expansion, the new node's state would be the accumulated thought steps from the path to the root. We show an example of the accumulated thought steps in the Appendix C.5.

Evaluation The evaluation phase in MCTS estimates the probability that a given node will suc-

cessfully complete the task. While some previous works refer to this as "simulation" (Zhou et al., 2023; Hao et al., 2023)—typically involving progression from intermediate to terminal states—we evaluate nodes by generating complete code based on the current thoughts and assessing this code's quality.

In code generation, a natural evaluation is to use the pass rate of public test cases (Zhang et al., 2023) as the reward. However, the limitation of this method is that public test cases cover only a part of the test set. When multiple code outputs pass all the public test cases, some may still fail to fully solve the problem, making it difficult to differentiate between them. To overcome this challenge, we propose a dual evaluation approach. Once all public test cases are passed, we further instruct the LLM to provide a self-assessed comprehensive score, v^{llm}, to evaluate the code's correctness in solving the whole problem.

$$\text{reward} = \begin{cases} v^{\text{test}}, & \text{if } 0 \leq v^{\text{test}} < 1 \\ a \times v^{\text{test}} + b \times v^{\text{llm}}, & \text{if } v^{\text{test}} = 1 \end{cases}, \tag{3}$$

where v^{test} is the pass rate on public test cases; v^{llm} is the LLM's self-evaluation score. a and b controls the weight of two parts.

The reward in this context is a scalar value, used to calculate the Q-value at each node and to determine the score during the selection phase. However, in code generation, the compiler and executor can return detailed error messages, and various code analysis tools can provide more granular insights into the code. These details about the code are crucial for making modifications but can not be captured in a scalar reward. Therefore, alongside the scalar reward, we also integrate verbal feedback.

Verbal Feedback When the generated code fails to pass a public test case, human programmers typically diagnose the issue by examining details such as variable values during execution. In the context of solving code generation tasks with search algorithms, relying solely on scalar feedback based on the pass rate of public test cases lacks detailed information. Therefore, we incorporate verbal feedback in the MCTS process. Specifically, as described in Sec. 3.2, we perform block-level analysis when the code fails a public test case and store the resulting information as verbal feedback in the current node. This feedback is then utilized in both the *expansion* and *rethink* phases.

Backpropagation In MCTS, backpropagation refers to the process of updating the Q-values of all nodes along the path from the current node to the root node using the rewards obtained from the evaluation. Beyond using scalar feedback to update the values of parent nodes, verbal feedback is also stored in the current leaf node for use in subsequent *expansion* and *rethink* phases.

Rethink When the code fails to pass a public test case, we can obtain block-level analysis as detailed verbal feedback on the execution. How can we leverage such fine-grained feedback to produce correct code? We propose to use this feedback to make the LLM "rethink", meaning to regenerate the current erroneous thought based on the feedback to avoid generating the incorrect code. As shown in Figure 2, the leaf node is re-generated by $z^{\text{new}} \sim p(z|s, f, z^{\text{old}})$. It is important to emphasize that we do not regenerate the parent nodes in the trace for two key reasons: 1) The parent nodes have already accumulated rewards over multiple rounds of evaluation from all their child nodes, and regenerating them would invalidate the previously gathered rewards. 2) The parent node has already gone through its own *rethink* process. This means that either the parent node did not encounter failing public test cases during its evaluation or has already been refined through the *rethink* process.

The advantage of introducing *rethink* is twofold. From the code generation perspective, *rethink* refines the reasoning process behind writing code, thus would ultimately lead to better code. From the MCTS perspective, it refines the current action or current node. Since the MCTS tree is built incrementally, improving the quality of the current action allows the LLM to explore more optimal paths in the vast search space, thereby enhancing the overall search quality of the tree. Through the *rethink* mechanism, we seamlessly integrate the process of refining the reasoning of code generation with the MCTS search process.

5 Experiment Settings

Datasets We evaluate RETHINKMCTS and baseline methods on two widely used benchmark datasets: APPS (Hendrycks et al., 2021) and HumanEval (Chen et al., 2021). The APPS dataset is a huge dataset contains three levels of difficulties: introductory, interview, and competition. Within

		Pass Rate (%)				Pass@1 (%)				
		APPS Intro.	APPS Inter.	APPS Comp.	Average	APPS Intro.	APPS Inter.	APPS Comp.	HumanEval	Average
GPT-3.5-turbo	Base(1)	50.43	40.57	23.67	38.22	29	19	9	70.12	37.07
	Base(16)	66.77	62.65	25.5	51.64	45	34	9	81.71	47.84
	PG-TD	60.89	50.80	26.50	46.06	40	25	8	76.22	42.67
	ToT	62.56	57.97	28.00	49.51	38	25	10	76.22	42.67
	LATS	54.06	45.86	21.83	40.58	36	20	7	79.88	41.81
	RAP	43.22	43.32	22.83	36.46	21	14	8	71.95	34.69
	LDB	56.68	46.78	21.00	41.49	35	22	8	81.09	42.67
	Reflexion	53.20	45.58	17.50	38.76	35	21	7	71.95	39.00
	RETHINKMCTS	**67.09**	**68.65**	**29.50**	**55.08**	**45**	**38**	**13**	**89.02**	**52.15**
GPT-4o-mini	Base(1)	56.56	52.40	35.00	47.98	35	29	16	87.20	48.06
	Base(16)	67.79	66.25	38.5	57.51	47	41	21	93.29	56.46
	PG-TD	66.97	67.15	39.83	57.98	47	43	23	91.46	56.68
	ToT	71.03	67.84	37.17	58.08	52	46	23	92.68	58.84
	LATS	69.46	67.65	35.83	57.65	50	45	19	93.29	57.54
	RAP	64.24	57.25	37.67	53.05	39	32	20	87.20	50.43
	LDB	60.64	60.78	40.33	53.91	40	38	23	90.85	53.87
	Reflexion	60.65	56.87	38.00	51.84	40	31	18	90.85	51.29
	RETHINKMCTS	**76.60**	**74.35**	**42.50**	**64.48**	**59**	**49**	**28**	**94.51**	**62.93**

Table 1: Performances of RETHINKMCTS and baselines on APPS and HumanEval. RETHINKMCTS achieves the best performance across all the datasets with the maximum number of rollouts for tree search algorithms being 16.

each difficulty, the problems are randomly distributed. Therefore, we elected the first 100 problems per difficulty to maintain randomness while ensuring balanced coverage, which mirrors sampling methods used by Zhang et al. (2023). We use *pass rate* and *pass@1* as the evaluation metrics for code correctness following (Zhang et al., 2023). *Pass rate* is the average percentage of private test cases successfully passed by the generated code across all problems, and *pass@1* measures the percentage of problems where the generated programs pass all private test cases (Austin et al., 2021; Chen et al., 2021; Dong et al., 2023).

Baselines To illustrate the effectiveness of RETHINKMCTS, we compare two kinds of code generation methods. The first kind is feedback-enhanced, which uses the code execution feedback to refine codes 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) and RAP (Hao et al., 2023). More details can be found in Appendix A.

Implementation We pick GPT-3.5-turbo and GPT-4o-mini as the backbone models to compare different algorithms. For search-enhanced methods, including RETHINKMCTS, we set the maximum number of children of any node to be 3. For MCTS-based methods, we set the hyperparameters in Eq. (2) c_{base} to be 10 and c to be 4 following previous work by Zhang et al. (2023). And we set the a and b in Eq. (3) to be $(0.8, 0.2)$ and we compare performances under different settings in Sec. 6. We set the maximum number of rollouts or simulation times to be 16. For LDB, we set the maximum number of debug times to be 10, as in the original paper (Zhong et al., 2024).

6 Results And Analysis

Overall Performance We present the overall performance in Table 1, where we can see that RETHINKMCTS outperforms all baseline models across both datasets. Additionally, by comparing them with the original base model, both feedback-enhanced and tree search-enhanced methods show significant performance improvements, demonstrating the effectiveness of exploring different strategies and using detailed feedback from code execution. Generally, RETHINKMCTS enhances performance more significantly on GPT-3.5-turbo than on GPT-4o-mini. This may be because weaker code models benefit more from error correction in the thought process.

Ablation Study We conduct ablation studies to remove each of our model's components, including self-evaluation (w/o selfEval), block-level analysis (partial verbal feedback, w/o blockInfo), whole verbal feedback (w/o VF), rethink mechanism (w/o *rethink*). The results using GPT-3.5-turbo as the backbone model are shown in Figure 3, and the results on GPT-4o-mini are presented in Appendix B. The results demonstrate that each component contributes to overall performance, with verbal feedback showing the most significant impact. This aligns with our design, as the *rethink* mechanism depends primarily on execution feedback—without this feedback, the model lacks the necessary information to refine its reasoning effectively.

Additionally, we can see that for the HumanEval

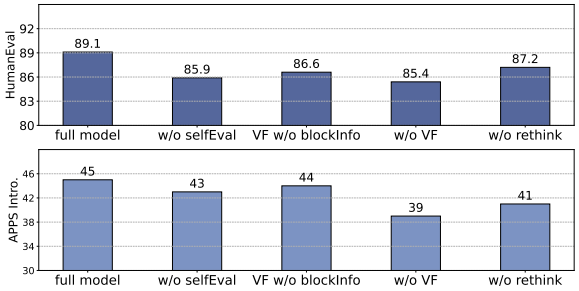

Figure 3: Ablation study of block-level analysis (blockInfo), rethink mechanism, verbal feedback (VF), and self-evaluation with GPT-3.5-turbo as the backbone.

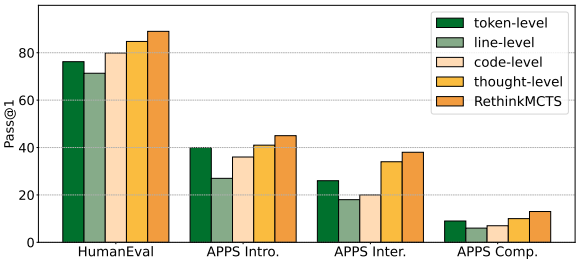

Figure 4: Performance comparison between different search granularity. For advanced models like GPT-3.5-turbo, it's better to explore at the thought level.

dataset, block-level analysis significantly impacts performance (89.1 → 86.6), while its effect on APPS is minimal. We attribute this to HumanEval having fewer public test cases than APPS (2.8 vs. 27.52 on average), making detailed test case analysis essential for the *rethink* mechanism to correct errors in HumanEval. This explains why dual evaluation is critical for HumanEval - the limited test cases necessitate LLM-based code reevaluation. Finally, the *rethink* mechanism we proposed significantly enhances the results. This improvement stems from that *rethink* enabling the use of fine-grained block-level analysis in verbal feedback, effectively correcting logical errors in the reasoning process.

Search Granularity Study RETHINKMCTS conducts a thought-level search for code. Here, we compare the action spaces for MCTS, specifically examining 4 levels of search granularity: token, line, code, and thought. The experimental results with GPT-3.5-turbo as backbone are presented in Figure 4, and the results on GPT-4o-mini are presented in Appendix B.

As shown in the figure, the thought-level search is more effective in finding viable code. This demonstrates that for advanced LLMs like GPT-3.5-turbo, exploring the reasoning process is beneficial (Zhang et al., 2024b; Huang and Chang, 2022). Additionally, we observe that token-level searching performs better than line and code-level searching. This is due to the fact that with a limited number of search iterations, token-level searches allow fewer constraints on the early tokens, thus uncovering more possibilities compared to line and code-level searches. Finally, although thought-level search yields the best results among different granularity, its effectiveness is further enhanced in RETHINKMCTS by introducing detailed feedback and *rethink* mechanism, making the search over thoughts in the code generation process even more effective.

Rethink vs. Reflection In this section, we compare *rethink* and *reflection* approaches. Our comparison methodology maintains all other components of RETHINKMCTS unchanged, with the only difference being the replacement of *rethink* with *reflection*. The experimental results are presented in Table 2. As demonstrated, *rethink* not only improves search effectiveness compared to *reflection* but also significantly reduces token consumption. This efficiency stems from *rethink*'s ability to directly modify incorrect thought steps, whereas *reflection* continuously accumulates error history, leading to excessive token consumption. Furthermore, since erroneous steps remain in the search tree with *reflection*, subsequent searches may continue down incorrect paths, resulting in inferior search performance.

Dataset	Reflection (Pass@1)	Rethink (Pass@1)	Reflection (Avg. Token Cost)	Rethink (Avg. Token Cost)
APPS-Intro.	54	**59**(↑ **9.2**%)	177353	**143048**(↓ **19.3**%)
APPS-Inter.	45	**49**(↑ **8.9**%)	163494	**126648**(↓ **22.5**%)
APPS-Comp.	24	**28**(↑ **16.6**%)	189215	**182193**(↓ **3.7**%)
HumanEval	93.29	**94.51**(↑ **1.3**%)	57027	**36678**(↓ **35.7**%)

Table 2: Comparison between rethink and reflection-based MCTS approaches. We experiment on RETHINKMCTS with other parts remain the same and only replace *rethink* with *reflection*.

Test Time Scaling with *Rethink* The goal of *rethink* is to improve the search quality within the same number of rollouts. To validate the effectiveness of *rethink*, we compare the performance between increasing the number of *rethink* operations and increasing the number of rollouts without applying *rethink*, while keeping the total number of rollouts consistent. The results are shown in Figure 5.

The figure shows that increasing the number of *rethink* operations and increasing the number of rollouts both enhance performance. This is ex-

(a, b)	Pass Rate (%)			Pass@1 (%)			
	APPS Intro.	APPS Inter.	APPS Comp.	APPS Intro.	APPS Inter.	APPS Comp.	HumanEval
(0.8, 0.2)	76.6	74.3	42.5	59	49	**28**	**94.5**
(1.0, 0.2)	76.9	**76.4**	**43.5**	**60**	53	27	92.7
(1.0, 1.0)	**78.8**	75.2	40.5	**60**	**54**	24	91.5

Table 3: Performance comparison under different reward weights. The (1.0, 0.2) and (1.0, 1.0) configurations make the nodes that achieve a pass rate of 1.0 on public test cases receive a score higher than 1.0, whereas the (0.8, 0.2) configuration keeps all node evaluations between 0∼1.

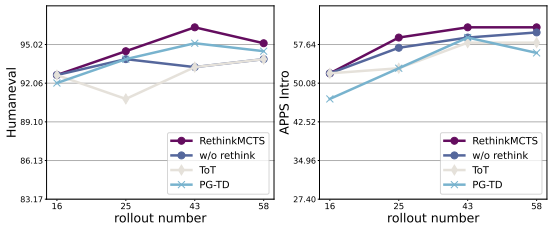

Figure 5: Performance comparison between *rethink* more times and more rollouts without *rethink*. *rethink* is more effective than increasing rollouts.

pected as more extensive exploration raises the probability of finding the correct code. However, increasing the number of *rethink* operations yields greater performance gains. This can be attributed to two key reasons. From a tree search perspective, without the *rethink* mechanism, erroneous actions or nodes would persist in the trace, causing the following nodes to follow incorrect reasoning paths, which makes it challenging to ensure the quality of the entire reasoning trace. From the code generation perspective, the *rethink* mechanism refines flawed thoughts and get a better thought chain, which would finally lead to better codes.

Method	APPS Intro.	HumanEval
W/O RETHINK	10.04	48.30
RETHINKMCTS	**15.60**	**53.29**

Table 4: The success rate comparison of the searched codes between with and without the *rethink* mechanism.

Furthermore, we compare the pass rate on public test cases of all the generated codes for the entire tree, with and without the *rethink* operation, since only public test cases are available during the search. The results are presented in Table 4. We can see that the *rethink* operation increases the proportion of effective code found in the tree. This highlights how refining erroneous thoughts enables the tree to focus more on correct paths, leading to better outcomes.

Study on Reward Weights We analyzed the impact of reward weights in Eq. (3) of Sec. 4, with results shown in Table 3. It is evident that (a, b) significantly influences RETHINKMCTS's performance, underscoring the importance of LLM self-evaluation. Since self-evaluation rewards apply only when code achieves a perfect pass rate on public test cases, each configuration yields distinct implications.

Under the (0.8, 0.2) configuration, the code is given a baseline score of 0.8, and the LLM's evaluation score is used to distinguish between different codes. This allows for situations where the total score of code that passes all public test cases could be lower than that of code with a pass rate below 1, but only when the LLM's self-evaluation score is particularly low. Conversely, configurations (1.0, 0.2) and (1.0, 1.0) ensure that code passing all public tests always receives a score exceeding 1.0. While this approach guarantees that final output maintains perfect public test performance, it prematurely discards promising reasoning paths with imperfect test results. This limitation explains the poorer performance observed on both datasets under these configurations.

7 Conclusion

We propose RETHINKMCTS, the first framework that searches and refines thoughts for code generation. Unlike previous tree search methods, RETHINKMCTS explores the reasoning process and incorporates an iterative *rethink* mechanism to improve search quality. Compared to traditional *reflection*, *rethink* achieves superior results with lower token cost by guiding search along correct paths. Experiments on APPS and HumanEval datasets demonstrate that RETHINKMCTS outperforms existing approaches, generating high-quality code through search-and-refinement reasoning. Beyond code generation, RETHINKMCTS offers a general approach for enhancing task performance through structured reasoning, with potential applications in other LLM domains such as mathematical problem-solving and tool usage scenarios.

Limitations

Limited Exploration of Fine-Tuning based on Collected Data RETHINKMCTS framework generates high-quality data that includes thought steps, execution feedback, and code. This data could potentially be used to fine-tune an LLM to enhance its code generation capabilities. However, since our work focuses primarily on the inference framework, we leave the fine-tuning exploration for future work.

Generalization to Other Reasoning Tasks Our primary contribution lies in developing a search framework that integrates code execution feedback for refinement. While this approach is effective for code generation tasks, it may not generalize well to other reasoning domains, such as mathematical reasoning, where similarly detailed feedback mechanisms might not be available. Nevertheless, our method could potentially be applicable to reasoning tasks that involve detailed feedback mechanisms comparable to those in code generation.

Potential Limitations in the Refinement Step Our current framework enhances the original MCTS approach by introducing a refinement step that utilizes detailed feedback from code execution. However, we do not introduce a sophisticated procedure for determining which specific thought step should be refined. Instead, we directly select the most recent step that caused the error, since in MCTS, the tree develops incrementally, ensuring each step eventually has the opportunity to be refined if it produces incorrect code. Although this approach proves effective, integrating a dedicated verifier to identify which thought step requires refinement could potentially yield better results.

Ethics Statement

In this work, we employ LLMs as both thought and code generators. All the dataset we use are publicly available and are for research purposes only. The LLMs utilized in our study include the closed-source model GPT-4o-mini and GPT-3.5-Turbo. Ethical considerations related to these models, including their training data and deployment, are addressed by their respective creators. The LLMs in our work are instructed solely to output code task-related thought steps, evaluation scores and code, and do not generate other 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

V Aho Alfred, S Lam Monica, and D Ullman Jeffrey. 2007. *Compilers principles, techniques & tools*. pearson Education.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and 1 others. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

Zhangqian Bi, Yao Wan, Zheng Wang, Hongyu Zhang, Batu Guan, Fangxin Lu, Zili Zhang, Yulei Sui, Hai Jin, and Xuanhua Shi. 2024. Iterative refinement of project-level code context for precise code generation with compiler feedback. *arXiv preprint arXiv:2403.16792*.

Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. 2012. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1):1–43.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, and 1 others. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Fanny Chevalier, David Auber, and Alexandru Telea. 2007. Structural analysis and visualization of c++ code evolution using syntax trees. In *Ninth international workshop on Principles of software evolution: in conjunction with the 6th ESEC/FSE joint meeting*, pages 90–97.

Michelle Cook, Megan Fowler, Jason O Hallstrom, Joseph E Hollingsworth, Tim Schwab, Yu-Shan Sun, and Murali Sitaraman. 2018. Where exactly are the difficulties in reasoning logically about code? experimentation with an online system. In *Proceedings of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education*, pages 39–44.

Matthew DeLorenzo, Animesh Basak Chowdhury, Vasudev Gohil, Shailja Thakur, Ramesh Karri, Siddharth Garg, and Jeyavijayan Rajendran. 2024. Make every move count: Llm-based high-quality rtl code generation using mcts. *arXiv preprint arXiv:2402.03289*.

Yihong Dong, Jiazheng Ding, Xue Jiang, Ge Li, Zhuo Li, and Zhi Jin. 2023. Codescore: Evaluating code generation by learning code execution. *arXiv preprint arXiv:2301.09043*.

Xidong Feng, Ziyu Wan, Muning Wen, Ying Wen, Weinan Zhang, and Jun Wang. 2023. Alphazero-like tree-search can guide large language model decoding and training. *arXiv preprint arXiv:2309.17179*.

Data Flow. 1994. *Control Flow Analysis*. Ph.D. thesis, QUEENSLAND UNIVERSITY OF TECHNOLOGY.

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999*.

Linyuan Gong, Mostafa Elhoushi, and Alvin Cheung. 2024. Ast-t5: Structure-aware pretraining for code generation and understanding. *arXiv preprint arXiv:2401.03003*.

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*.

Zhenyu He, Jun Zhang, Shengjie Luo, Jingjing Xu, Zhi Zhang, and Di He. 2024. Let the code llm edit itself when you edit the code. *arXiv preprint arXiv:2407.03157*.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and 1 others. 2021. Measuring coding challenge competence with apps. *arXiv preprint arXiv:2105.09938*.

Zhiyuan Hu, Chumin Liu, Xidong Feng, Yilun Zhao, See-Kiong Ng, Anh Tuan Luu, Junxian He, Pang Wei Koh, and Bryan Hooi. 2024. Uncertainty of thoughts: Uncertainty-aware planning enhances information seeking in large language models. *arXiv preprint arXiv:2402.03271*.

Dong Huang, Qingwen Bu, Jie Zhang, Xiaofei Xie, Junjie Chen, and Heming Cui. 2023a. Bias assessment and mitigation in llm-based code generation. *arXiv preprint arXiv:2309.14345*.

Dong Huang, Qingwen Bu, Jie M Zhang, Michael Luck, and Heming Cui. 2023b. Agentcoder: Multi-agent-based code generation with iterative testing and optimisation. *arXiv preprint arXiv:2312.13010*.

Jie Huang and Kevin Chen-Chuan Chang. 2022. Towards reasoning in large language models: A survey. *arXiv preprint arXiv:2212.10403*.

Yoichi Ishibashi and Yoshimasa Nishimura. 2024. Self-organized agents: A llm multi-agent framework toward ultra large-scale code generation and optimization. *arXiv preprint arXiv:2404.02183*.

Haolin Jin, Linghan Huang, Haipeng Cai, Jun Yan, Bo Li, and Huaming Chen. 2024. From llms to llm-based agents for software engineering: A survey of current, challenges and future. *arXiv preprint arXiv:2408.02479*.

Sumith Kulal, Panupong Pasupat, Kartik Chandra, Mina Lee, Oded Padon, Alex Aiken, and Percy S Liang. 2019. Spoc: Search-based pseudocode to code. *Advances in Neural Information Processing Systems*, 32.

James R Larus. 1999. Whole program paths. *ACM SIGPLAN Notices*, 34(5):259–269.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, and 1 others. 2023. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*.

Yichen Li, Yun Peng, Yintong Huo, and Michael R Lyu. 2024. Enhancing llm-based coding tools through native integration of ide-derived static context. In *Proceedings of the 1st International Workshop on Large Language Models for Code*, pages 70–74.

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*.

Silin Meng, Yiwei Wang, Cheng-Fu Yang, Nanyun Peng, and Kai-Wei Chang. 2024. Llm-a*: Large language model enhanced incremental heuristic search on path planning. *arXiv preprint arXiv:2407.02511*.

Daye Nam, Andrew Macvean, Vincent Hellendoorn, Bogdan Vasilescu, and Brad Myers. 2024. Using an llm to help with code understanding. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, and 1 others. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

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, and 1 others. 2017. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*.

Xiaojuan Tang, Zilong Zheng, Jiaqi Li, Fanxu Meng, Song-Chun Zhu, Yitao Liang, and Muhan Zhang. 2023. Large language models are in-context semantic reasoners rather than symbolic reasoners. *arXiv preprint arXiv:2305.14825*.

Ante Wang, Linfeng Song, Ye Tian, Baolin Peng, Dian Yu, Haitao Mi, Jinsong Su, and Dong Yu. 2024a. Litesearch: Efficacious tree search for llm. *arXiv preprint arXiv:2407.00320*.

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

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, and 1 others. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837.

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.

Lifan Yuan, Ganqu Cui, Hanbin Wang, Ning Ding, Xingyao Wang, Jia Deng, Boji Shan, Huimin Chen, Ruobing Xie, Yankai Lin, and 1 others. 2024. Advancing llm reasoning generalists with preference trees. *arXiv preprint arXiv:2404.02078*.

Kechi Zhang, Jia Li, Ge Li, Xianjie Shi, and Zhi Jin. 2024a. Codeagent: Enhancing code generation with tool-integrated agent systems for real-world repo-level coding challenges. *arXiv preprint arXiv:2401.07339*.

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*.

Yadong Zhang, Shaoguang Mao, Tao Ge, Xun Wang, Adrian de Wynter, Yan Xia, Wenshan Wu, Ting Song, Man Lan, and Furu Wei. 2024b. Llm as a mastermind: A survey of strategic reasoning with large language models. *arXiv preprint arXiv:2404.01230*.

Lin Zheng, Jianbo Yuan, Zhi Zhang, Hongxia Yang, and Lingpeng Kong. 2023. Self-infilling code generation. In *Forty-first International Conference on Machine Learning*.

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*.

Changzhi Zhou, Xinyu Zhang, Dandan Song, Xiancai Chen, Wanli Gu, Huipeng Ma, Yuhang Tian, Mengdi Zhang, and Linmei Hu. 2025. Refinecoder: Iterative improving of large language models via adaptive critique refinement for code generation. *arXiv preprint arXiv:2502.09183*.

Appendix

A Details of Baselines

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.

B Additional results

This section presents some additional experiment results.

Ablation Study Here, we present the results of the ablation study using GPT-4o-mini as the backbone model, as shown in Figure 6. It is clear that the *rethink* operation and verbal feedback remain the most significant contributors to our model's performance. Notably, the *rethink* mechanism exhibits even stronger effects with GPT-4o-mini than with GPT-3.5-turbo, likely due to the model's enhanced ability to effectively utilize feedback and make refinement.

Search Granularity Study We present the results of the search granularity study using GPT-4o-mini as the backbone model, shown in Figure 7. It is evident that the differences across granularities are smaller on the HumanEval dataset, likely due to its relatively low overall difficulty. However, on the APPS dataset, the advantage of thought-level search becomes much more pronounced, especially at the highest "competition" difficulty level. This suggests that for more complex problems, exploring the thought process and reasoning is beneficial.

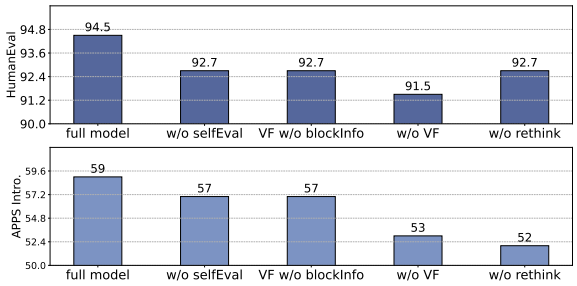

Figure 6: Ablation study of block-level analysis (blockInfo), rethink mechanism, the verbal feedback (VF) and self-evaluation with GPT-4o-mini as the backbone.

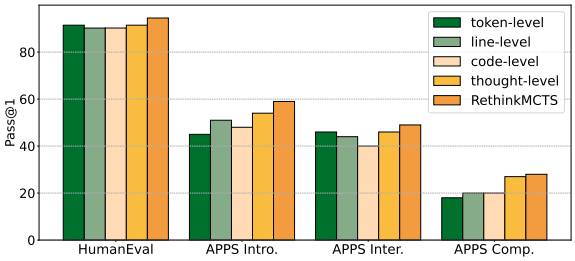

Figure 7: Performance comparison between different search granularity. For advanced model like GPT-4o-mini, it's better to explore at the thought level.

Token Consumption Our approach ensures the comparison fair by keeping the number of rollouts, i.e., the number of codes generated during the search, the same. This is following the previous work about tree search in code generation PG-TD (Zhang et al., 2023) and LATS (Zhou et al., 2023).

However, one limitation is that our method would cost more tokens since we have introduce block-level analysis and rethink mechanism. Here we present the detailed token usage on GPT-4o-mini in Table 5.

Our increase in token usage is primarily due to the introduction of block-level analysis, which includes the values of variables before and after each block. These values are obtained through code execution, resulting in longer textual inputs to the LLM. However, the feedback makes our model deliver significantly better results. It is essential to the success of the "rethink" mechanism, and it represents an important characteristic in the coding environment (there would be no such detailed feedback for the math reasoning problem). Therefore, incorporating such feedback is crucial. Like OpenAI's o1 model (which solves one problem with thousands of tokens in the hidden CoT and minutes to take), our primary aim is not to optimize token count or computation time. Instead, the emphasis is on enabling LLMs to generate higher-quality reasoning processes and achieve superior reasoning outcomes.

Self-evaluation vs. Self-generating Unit Tests Given the limited coverage of public test cases, we propose a dual evaluation approach. In this section, we compare it with an alternative approach of self-generating unit tests. In the latter approach, when the code passes the public test cases, we have the LLM generate additional test cases and get a new *pass rate* on these tests. The combined results serve as a comprehensive evaluation of the code. Experimental results are shown in Table 6.

As the table demonstrates, while self-generating unit tests improve the *pass rate* on test cases, they do not improve the *pass@1* metric. This is because self-evaluation directly assesses the code after it passes the public test cases, scoring it based on how well it meets the problem's requirements. As a result, it provides a more accurate indication of the code's ability to address the entire problem. In contrast, self-generating unit tests focus on creating additional tests, which emphasize the test suite rather than the code itself. There are two potential reasons for this: 1) Self-generating unit tests primarily identify patterns in the existing tests and generate a set of tests that better match the test suite. This can enhance the *pass rate* by filtering for code that matches these patterns, but it doesn't necessarily identify the mismatch between the code and the problem requirement. 2) The generated tests may not always be correct (Huang et al., 2023b), which can mislead the code's modification process and the subsequent search direction, potentially steering it away from valid solutions.

Multiple Runs To further illustrate our method's advantage, we run our method and strong baselines for 3 times with different random seeds. The average performance and the standard derivation

	APPS Intro. (%)			HumanEval		
	#Input token	#Output token	Cost($)	#Input token	#Output token	Cost($)
ToT	24799	7156	0.008	11687	7131	0.006
LATS	104634	17472	0.026	12690	7403	0.006
PG-TD	27827	5378	0.007	5959	3759	0.003
LDB	61112	1734	0.010	13161	480	0.002
RETHINKMCTS	123207	17863	0.029	28479	8198	0.009

Table 5: The token consumption comparison. The results represent the average number of tokens consumed per question.

	Pass Rate (%)			Pass@1 (%)			
	APPS Intro.	APPS Inter.	APPS Comp.	APPS Intro.	APPS Inter.	APPS Comp.	HumanEval
Direct Evaluation	76.60	74.34	42.50	59	49	28	94.51
Self-generated Tests	77.32	75.80	47.23	59	44	28	93.29

Table 6: The performance comparison between using Direct Self-evaluation and Self-generating test evaluation.

are presented in Table 7. Noticeably, our model, RETHINKMCTS, consistently demonstrates a stable performance advantage across multiple experiments. Additionally, we have noticed that since we set the temperature to 0, the standard deviation between different runs is very small. Therefore, we included the result of only one run in the main body of the text.

C Prompts

In this section, we present the prompts used when an LLM to perform various operations.

C.1 Expansion Prompt

First, we discuss the prompts for the Expansion step in the MCTS process. There are two sets of prompts: one set is used to generate new thoughts based on the problem description and previous thoughts when there is no feedback presented in Table 8;

The other set is used when the generated code contains errors and verbal feedback is provided. In this case, the LLM uses the verbal feedback to generate thoughts that avoid such errors. We present the prompt in Table 9.

C.2 Code Generation Prompt

We present the prompt we use to instruct the LLM to generate code following previous thoughts in Table 10.

C.3 Evaluation Prompt

Besides the normal evaluation on the public test cases, we also develop an LLM-based self-evaluation when the public test cases are all passed. Here we present the prompts in Table 11.

C.4 Rethink Prompt

When the generated code following some thoughts doesn't pass some public test cases, we would use the block-level analysis to form the verbal feedback and use it to refine the previous thought, a.k.a, rethink. Here we present the prompt for this operation in Table 12.

C.5 An Example of Accumulated Thoughts

Here we present an example of the thought steps accumulated in one trace of MCTS tree in Table 13.

C.6 An Example of Verbal Feedback

The verbal feedback we constructed contains the detailed block-level analysis of the code. Here we present an example of it.

D Algorithm

We present the detailed procedure of RETHINKMCTS in pseudocode in Algorithm 1.

	Pass Rate (%)			Pass@1 (%)			
	APPS Intro.	**APPS Inter.**	**APPS Comp.**	**APPS Intro.**	**APPS Inter.**	**APPS Comp.**	**HumanEval**
ToT	72.22±1.19	67.10±1.06	40.33±2.58	53.67±1.70	45.00±0.82	22.67±2.05	92.48±0.29
LATS	70.17±0.52	68.66±0.79	36.83±3.61	50.33±0.47	45.67±0.94	18.33±3.30	93.70±0.57
PG-TD	68.85±2.29	68.29±1.48	40.33±2.00	49.00±4.32	44.33±1.25	23.00±1.63	92.28±0.76
RETHINKMCTS	**75.36**±1.08	**74.10**±0.98	**43.33**±1.18	**57.33**±1.25	**50.00**±0.82	**27.00**±0.82	**94.31**±0.29

Table 7: Comparing RETHINKMCTS with competitive baselines in multiple runs. The mean and standard deviation of the results are presented.

Prompt for Rethink

```
{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 8: Prompt for generating thoughts in search methods.

Prompt for Rethink

```
{problem statement}

{thoughts}
```python
generated code
```
{verbal feedback}
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.
* The goal is that the thoughts could lead to the code that not only avoids the current error but also solve the problem in a way that handles other potential test cases that we haven't encountered yet.
* 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 9: Prompt for generating thoughts in search methods.

Prompt for Rethink

```
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 10: Prompt for generating the code following the thoughts in search methods.

Prompt for Rethink

```
{problem statement}

{thoughts}
```python
generated code
```

Above is a Python code problem with the thoughts and code to solve the problem. The code could pass all the example test cases, however, it may or may not be completely correct.

Please evaluate and return the correctness score in range [-1, 1].

Evaluate the correctness of the code and give only ONE evaluation score.

The code's correctness is whether it can pass all the possible unseen test cases of the problem, not just the given ones.

Example Answers:
{"evaluation": -0.5, "explanation": The code is far from correct for solving the problem.}
{"evaluation": 0.1, "explanation": The code is not the correct solution but can pass some simple test cases.}
{"evaluation": 0.85, "explanation": The code can pass most test cases while may fail on some corner cases.}
{"evaluation": 1.0, "explanation": The generated code is the correct solution that can pass all the possible test cases and strange corner cases too.}
```

Table 11: Prompt for evaluating the thoughts and code.

Prompt for Rethink

```
{problem statement}
{thoughts}
```python
generated code
```
{verbal feedback}

Based on your previous thoughts and the new experience, please provide a new Thought to replace the previous thought. This new thought should avoid the mistake.

Remember that you only need to provide the thought (one or two sentences) to solve the problem, not the code.
```

Table 12: Prompt for Rethink

An Example of Accumulated Thoughts

```
1-Thought:To maximize the number of edges covered by paths between three distinct
    ↪ vertices in a tree, we can ... yield a higher number of unique edges.

2-Thought:To enhance the previous thought about leveraging the diameter of the tree, we
    ↪ can implement a breadth-first search (BFS) twice: ... maximizes unique edges
    ↪ covered when chosen as the third vertex.

3-Thought:To maximize the number of edges covered by paths between three distinct
    ↪ vertices in a tree, we can implement a two-pass BFS ... to maximize edge coverage.

4-Thought:To further enhance the previous thoughts on maximizing edge coverage among
    ↪ three vertices in a tree, ..., thus ensuring we maximize the unique edges covered
    ↪ by the paths connecting the three vertices.
```

Table 13: An example of accumulated thought steps. The thoughts are from GPT-4o-mini when solving APPS problem 4000 (1st problem of introductory difficulty):

An Example of Verbal Feedback

```
(Problem and Code)
import math
def largest_divisor(n: int) -> int:
    """ For a given number n, find the largest number that divides n evenly, smaller than
    ↪ n
    >>> largest_divisor(15)
    5
    """
    for i in range(int(math.sqrt(n)), 0, -1):
        if n % i == 0 and i < n:
            return i
    return 1  # In case n is 1, return 1 as the largest divisor

(Execution Feedback)
The code above fails the given unit test:
assert largest_divisor(15) == 5
# Current Execution Output:
3.

(Block-level Analysis)
Here is the code execution trace block by block with the intermediate variable values.
    ↪ Please explain the execution FOR EACH BLOCK and answer whether this block is
    ↪ correct or not. If not, give an explanation on what is wrong. Please wrap your
    ↪ response into a JSON object that contains keys `block` with the name of each
    ↪ block, key `correct` with value False or True, and key `explanation` with an
    ↪ explanation on the bug.
Example Answers:
{"block": "BLOCK-1", "correct": "True", "explanation": "The block initializes variable
    ↪ `a` and `b`."}
{"block": "BLOCK-2", "correct": "False", "explanation": "The block is incorrect because
    ↪ the code does not add the two integers together, but instead subtracts the second
    ↪ integer from the first. To fix this issue, we should change the operator from `-`
    ↪ to `+` in the return statement. This will ensure that the function returns the
    ↪ correct output for the given input."}
[BLOCK-0]
    # n=15
    for i in range(int(math.sqrt(n)), 0, -1):
    # n=15  i=3
[BLOCK-1]
        # n=15  i=3
        if n % i == 0 and i < n:
        # n=15  i=3
[BLOCK-2]
            # n=15  i=3
            return i
            # n=15  i=3 _ret=3

# LLM-analysis
```json
[
    {"block": "BLOCK-0", "correct": "True", "explanation": "The block initializes the
    ↪ loop with the correct starting point, which is the integer part of the square root
    ↪ of n."},
    {"block": "BLOCK-1", "correct": "False", "explanation": "The block checks if 3 is a
    ↪ divisor of 15, which is correct, but we are not immediately returning the largest
    ↪ divisor. The loop continues, and the next iteration will check smaller values of
    ↪ i."},
    {"block": "BLOCK-2", "correct": "False", "explanation": "The block incorrectly
    ↪ returns 3 instead of the larger valid divisor. Since the loop searches from the
    ↪ square root downwards, it should continue checking until it finds the next valid
    ↪ divisor, which is 5. The return statement needs to occur outside of this loop and
    ↪ only when the largest divisor is found."}
]
```
```

Table 14: An example of verbal feedback.

Algorithm 1 The RETHINKMCTS algorithm.

Require: $root$: the problem description; c: P-UCB exploration parameter; k: the maximum number of children of any node; a, b: the reward weights of the pass rate and the LLM evaluation.

$program\_dict = \texttt{DICTIONARY}()$
$verbal\ feedback\ f = \texttt{EMPTY}$
for $i \leftarrow 1, 2, \ldots, max\_rollouts$ **do**
  $node \leftarrow root$
  # Selection
  while $|node.children| > 0$ **do**
    $node \leftarrow \texttt{P\_UCB\_SELECT}(node.children, c)$
  end while
  # Expansion
  $next\_thoughts \leftarrow \texttt{TOP\_K}(node, k)$
  for $next\_thought \in next\_thoughts$ **do**
    $next\_state \leftarrow \texttt{CONCAT}(node, next\_thought)$
    Create a node new_node for $next\_state$
    Add new_node to the children of $node$
  end for
  # Evaluation
  $C \leftarrow \texttt{GENERATE}(node)$
  $v^{\text{test}}, f \leftarrow \texttt{GET\_PASS\_RATE}(p)$
  $v^{\text{llm}}, f \leftarrow \texttt{GET\_LLM\_EVAL}(p)$
  $program\_dict[C] = r = a * v^{\text{test}} + b * v^{\text{llm}}$
  if $v^{\text{test}} = 1$ **then**
    $program\_dict[C] = r = a * v^{\text{test}} + b * v^{\text{llm}}$
  else
    $program\_dict[C] = r = v^{\text{test}}$
  end if
  # Backpropagation
  Update and the values of $node$ and its ancestors in the tree with r
  # Rethink
  if $v^{\text{test}} \neq 1$ **then**
    $node.thought = \texttt{RETHINK}(node, f)$
    $next\_thoughts = \texttt{RETHINK\_NEXT}(node, k, f)$
    $C = \texttt{RE-GENERATE}(node)$
    $r = \texttt{RE-EVALUATION}(C)$
    $program\_dict[C] = r$
  end if
end for
return program in $program\_dict$ with the highest reward