RETHINKMCTS: REFINING ERRONEOUS THOUGHTS IN MONTE CARLO TREE SEARCH FOR CODE GENERATION

Qingyao Li 1 Wei Xia 2 Kounianhua Du 1 Xinyi Dai 2 Ruiming Tang 2 Yasheng Wang 2 Yong Yu 1 Weinan Zhang 1 Shanghai Jiao Tong University 2 Huawei Noah's Ark Lab $\{1y890306, wnzhang\}$ @sjtu.edu.cn

ABSTRACT

LLM agents enhanced by tree search algorithms have yielded notable performances in code generation. However, current search algorithms in this domain suffer from low search quality due to several reasons: 1) Ineffective design of the search space for the high-reasoning demands of code generation tasks, 2) Inadequate integration of code feedback with the search algorithm, and 3) Poor handling of negative feedback during the search, leading to reduced search efficiency and quality. To address these challenges, we propose to search for the reasoning process of the code and use the detailed feedback of code execution to refine erroneous thoughts during the search. In this paper, we introduce RethinkMCTS, which employs the Monte Carlo Tree Search (MCTS) algorithm to conduct thought-level searches before generating code, thereby exploring a wider range of strategies. More importantly, we construct verbal feedback from finegrained code execution feedback to refine erroneous thoughts during the search. This ensures that the search progresses along the correct reasoning paths, thus improving the overall search quality of the tree by leveraging execution feedback. Through extensive experiments, we demonstrate that RethinkMCTS outperforms previous search-based and feedback-based code generation baselines. On the HumanEval dataset, it improves the pass@1 of GPT-3.5-turbo from 70.12 to 89.02 and GPT-40-mini from 87.20 to 94.51. It effectively conducts more thorough exploration through thought-level searches and enhances the search quality of the entire tree by incorporating *rethink* operation¹.

1 Introduction

Coding has become an increasingly valuable skill in the digital information era, serving as the language that controls electronic machines. As the capabilities of large language models (LLMs) continue to impress, research has increasingly focused on enhancing their code generation abilities. Early efforts concentrated on pre-training models specifically for code generation using vast amounts of code data. With the growing power of general LLMs and the need for external tools and resources such as compilers and code libraries in code generation tasks, utilizing general LLMs as agents to improve code generation through external feedback or algorithmic design has emerged as a promising direction.

In research where LLMs are used as agents for code generation, search methods have been widely applied and have demonstrated remarkable effectiveness. These methods often iteratively refine solutions while exploring various possibilities through search techniques. Despite achieving notable results, previous approaches still face three key challenges: 1) The lack of modeling and exploration of the reasoning process that precedes code writing. Previous studies on LLM reasoning capabilities, such as chain-of-thought (Wei et al., 2022) and tree of thoughts, have shown that explicitly representing the reasoning process is crucial for the success of reasoning tasks. However, prior work in code generation has not explicitly modeled the relationship between reasoning (thought)

¹Resources are available at https://github.com/SIMONLQY/RethinkMCTS.

and code. Given that code generation is a task that demands strong reasoning abilities, the foundational elements of reasoning, i.e., thoughts, have not been fully explored or utilized. 2) The failure to effectively integrate the feedback of code execution with search algorithms. Unlike other reasoning tasks, code generation often benefits from detailed feedback provided by compilers. In earlier search algorithms, this feedback was typically incorporated into the subsequent generation process simplistically, often by using memory to store feedback. However, this approach is rudimentary and does not effectively use feedback to identify and correct errors. Enhancing the reasoning process in code generation by leveraging the feedback code executions remains a critical challenge. 3) Poor handling of errors during search. Previous methods often employed two strategies when encountering poor evaluation results during the search process. The first strategy involved self-reflection (Zhou et al., 2023) to summarize experiences and incorporate these insights into the context of subsequent explorations. The second strategy was to directly prune the search tree to improve efficiency (Zhang et al., 2023; Yao et al., 2024). However, these approaches failed to enhance the quality of the search. Although self-reflection provided summaries of past errors, the erroneous actions remained in the exploration path, causing subsequent searches to continue along incorrect trajectories. On the other hand, while pruning improved efficiency, it also risked discarding potentially valuable paths.

In this paper, we propose a method to integrate search algorithms into the code generation process by exploring code reasoning pathways. We utilize fine-grained feedback from the coding environment to correct reasoning errors, thereby improving the overall quality of the search tree, leading to the proposal of the RethinkMCTS. Specifically, we propose a code generation and search framework based on MCTS for reasoning-to-coding. In this framework, we explicitly explore different coding strategies by leveraging the search properties of MCTS. More importantly, we propose a "rethink" operation that combines code execution feedback with the search algorithm, enabling the model to refine erroneous thoughts based on the feedback. This allows the search algorithm to continue exploring along the refined paths, thereby enhancing the overall quality of the search tree. Furthermore, to better guide the evaluation of actions during the MCTS search process, we introduce dual evaluation, which aims to ensure effective code selection, especially in situations where public test cases alone are insufficient.

It is worth noting that previous work in the field of LLM agents has explored self-reflection-based approaches (Zhou et al., 2023; Shinn et al., 2024). These approaches typically summarize past mistakes and store them as memories, which are then used as inputs for future reasoning or planning. However, they do not correct the erroneous reasoning or planning thoughts within the trace itself. In contrast, our work explicitly corrects errors in the thoughts during code writing, ensuring that the search proceeds along higher-quality traces. Additionally, we focus on establishing a relationship between the code and the preceding thoughts. Specifically, our main contributions are as follows:

- Reasoning-to-Code Search Framework for Code Generation: The framework employs a multistep thinking and single-step code generation approach using Monte Carlo Tree Search (MCTS) to explore various code generation strategies. A combination of verbal and scale feedback guides the MCTS tree generation. To the best of our knowledge, we are the first to try to search and refine the thought process of code to enhance LLMs on code generation.
- Incorporating Verbal Feedback to Refine Thought Errors in MCTS: Verbal feedback is introduced to MCTS to correct errors in the current thought process, enabling the correction of mistakes in the current trace rather than continuing along the erroneous path.
- Introducing Detailed Feedback and Dual Evaluation for Error Correction: Detailed feedback is used to pinpoint errors at the block level, guiding the correction of faulty thought. Additionally, a dual evaluation method—using both visible tests and LLM evaluations—is proposed. This ensures effective code selection, especially when visible tests alone are insufficient.

2 RELATED WORK

LLMs for Code Generation. Large language models (LLMs) have been widely applied and developed in the field of code due to their powerful reasoning capabilities. The work on LLMs for code generation can be broadly categorized into two types: the first type involves fine-tuning or pre-training LMs specifically on code data (Luo et al., 2023; Li et al., 2023; Fried et al., 2022; Roziere et al., 2023), which makes them to get a deep understanding of code syntax, semantics, and structures. Strong foundation models like GPT-3.5-turbo and GPT-4 have also been used for code

generations (Madaan et al., 2024). Another line of works takes LLMs as agents (Zhang et al., 2023; Huang et al., 2023a; Zhong et al., 2024). They usually first design an procedure of coding and make LLMs to play different roles in the procedure. LDB proposed by Zhong et al. (2024) takes the LLM as the debugger and using block-level decomposition to locate bugs, finally enhance the code generation performance. PG-TD proposed by Zhang et al. (2023) utilize Monte Carlo Tree Search (Browne et al., 2012) methods combine with the probabilistic output of LMs to achieve token-level search for code generation. Reflexion proposed by Shinn et al. (2024) takes the LLM to generate reasoning, action and reflections to make the LLM learn from the past experience, which achieved impressive performance on code generation problem. Although achieving great performances, these methods fail to build the process of from reasoning to coding and ignore the importance of using the feedback in coding environment to correct the error in the coding process, which is the focus of our work.

Tree Search-based Reasoning. Tree search methods can maximize the exploration capabilities of LLMs, allowing for exploration at different levels (Zhang et al., 2023; Hu et al., 2024; Hao et al., 2023). PG-TD (Zhang et al., 2023) leverages the probability output characteristics of tokens in the Transformer architecture of LMs to explore at the token level, achieving fine-grained search for the correct code. Tree of Thoughts (ToT) (Yao et al., 2024) builds on the Chain-of-Thoughts (CoT) (Wei et al., 2022) by breaking down the reasoning steps of a problem and exploring the thought for each step. LATS (Zhou et al., 2023) treats the LLM as a more general agent, exploring at both the reasoning and action levels. For code generation tasks, it directly explores at the entire code segment level and uses reflection to summarize past experiences, thereby enhancing the exploration of subsequent trajectories. TS-LLM (Feng et al., 2023) proposes a tuning-based method that learns a value function to guide the decoding process. However, we focus on tuning-free algorithms that enhance the code generation capabilities of LLMs in an off-the-shelf manner. All these methods have improved the reasoning capabilities of LLMs significantly. However, when it comes to code generation tasks, these methods may be less effective. Because direct search at the code or token level lacks modeling of the reasoning process. And, the feedback from the coding environment is not effectively integrated by them. In this paper, we focus on leveraging the detailed feedback from code execution environment to search and refining the thought process, thereby improving the quality of exploration.

3 Preliminaries

3.1 Problem Formulation

The task is code generation and we follow the setting in previous work. Zhang et al. (2023). Specifically, for a large language model (LLM) agent, the input is the problem statement P and a public test case set $T_{\rm pub}$. The aim is to build a code generation agent model M that can output the correct code $C \sim M(P, T_{\rm pub})$ for solving the problem. We assume the agent has access to a set of test cases, each consisting of an input-output pair. To evaluate the effectiveness of the generated code, in addition to the public test cases $T_{\rm pub}$, we also retain private test cases $T_{\rm priv}$. These private test cases are not visible to the agent during the code generation process. The primary metric for assessing the quality of the generated code is whether it can pass these private test cases.

3.2 BLOCK-LEVEL CODE ANALYSIS

Execute buggy code in executor can only get standard error information. If the code runs without errors but produces incorrect output, there is often no error information available. However, since code is highly structured, it is possible to obtain detailed execution feedback through structured analysis of the code execution. We follow previous work LDB by Zhong et al. (2024) to get a code analysis report at the block level. In the code static analysis area, 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 the graph to get an execution trace of this test case, $[B_1, B2, ..., B_n]$, where each node within the CFG corresponds to a basic block. Given the trace, we execute the blocks one by one to collect all variables' state changes following the trace. These variables' state changes are collected and given to the LLM to get the block-level analysis of whether each block is correct or wrong.

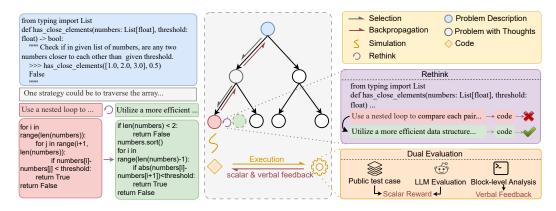


Figure 1: Overview of RethinkMCTS. We use MCTS to explore different thoughts of writing code. We obtain block-level analysis as verbal feedback through a code executor and use the verbal feedback from failed test cases to refine the thoughts, thereby improving the overall quality of the search tree.

3.3 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is a heuristic search algorithm that achieves great success in decision-making tasks (Silver et al., 2016). MCTS combines the precision of tree search with the randomness of Monte Carlo simulations to make decisions in complex environments. It initials the problem description as the root node and moves down the tree by selecting actions (child nodes) until the leaf node according to the Upper Confidence Bound (UCB) (Silver et al., 2017) algorithm that balances exploration and exploitation. Then, the MCTS would generate new child nodes for the chosen leaf node. For the newly generated node, the MCTS would simulate it until the terminal state and assign an evaluation reward to this node. Finally, the reward is backpropagated along the way back to the root node. Each node would update its value based on the newly collected reward.

4 RETHINKMCTS

Overview The motivation behind RethinkMCTS is to utilize feedback from the coding environment to refine the search process of the thoughts for writing code, ultimately identifying the correct solution. To achieve that, we take an LLM as an agent that can generate thoughts and strategies for writing codes and further generate codes based on the thoughts. We employ Monte Carlo Tree Search (MCTS) as the search algorithm to balance exploration and exploitation during the search for thoughts. More importantly, we introduce the "rethink" operation to leverage detailed feedback from code execution to refine erroneous thoughts, allowing the search to proceed along better paths and thereby improving the quality of the search. The framework is shown in Figure 1. 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.
- Block-Level Analysis Feedback: We use block-level analysis tools to obtain fine-grained feedback at the block level during code execution.
- **Rethink Mechanism**: We introduce a *rethink* mechanism that leverages feedback from the code execution to improve the quality of the reasoning process.
- Dual Evaluation: In the evaluation phase, we propose a dual evaluation method that combines
 public test cases and LLM evaluation to evaluate the generated codes, ultimately identifying highquality solutions.

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

Selection In MCTS, the selection step aims to balance exploration and exploitation by iteratively choosing actions most worthy of further expansion until a leaf node is reached. The node is selected based on a score calculated based on the visited times and stored value of the state and action pair (s,a), where the state s is the problem description and previous thoughts and action a is the new thought of this node. Each node would retain a Q(s,a) as the value, which is the maximum reward obtained by starting in s and taking action a. Here, we use P-UCB (Silver et al., 2017), an improved version of the UCB algorithm, to calculate the overall score for each node:

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

where $s^{'}$ is the state reached by taking action a in s; $p(a \mid s)$ 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

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

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

In each state, or at each node, the selection process will choose the action with the highest P-UCB value according to the formula described above, continuing this process until a leaf node is reached.

Expansion After selecting a leaf node, the expansion step aims to generate its child nodes, thereby extending the search scope of the entire tree. We define the action space for the search as the thoughts or strategies of writing the code. To utilize the feedback provided in the code environment, we handle 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 agent then proposes multiple subsequent thoughts z and gives a reasonableness score e of each thought as the p(a|s) in equation 1 based on the previous thoughts and 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 has passed all public test cases, the expansion step involves instructing the agent to propose subsequent thoughts without additional feedback, i.e., $[(z^1, e^1), \ldots, (z^k, e^k)] \sim p((z, e)^{(1 \cdots k)} | s)$.

Evaluation The primary purpose of the evaluation is to assess the likelihood that the current node will successfully complete the final task. Some previous works refer to the evaluation step of MCTS as "simulation" (Zhou et al., 2023; Hao et al., 2023) because it involves simulating from the intermediate state of a node to a terminal state to obtain a reward. For the task of code generation, we search for the thoughts and evaluate with the code generated following the thoughts, meaning we generate complete code based on the currently produced thoughts and use the evaluation of the code as the reward.

For code generation task, a natural approach is to use the pass rate on public test cases (Zhang et al., 2023) as the reward. However, the drawback of this method is that public test cases only cover part of the test space. When multiple codes pass all the public test cases, some of them may not be correct for solving the problem, but it becomes difficult to distinguish between them. Some previous efforts have tried to have LLMs generate more test cases to cover as many potential scenarios as possible (Huang et al., 2023a; Zhou et al., 2023). However, this approach is costly and does not guarantee the accuracy of the generated test cases. To address this challenge, we propose a dual evaluation approach. When the public test cases are all passed, we further instruct the LLM to provide a self-evaluated comprehensive score $v^{\rm llm}$ for the code's correctness to pass other potential test cases.

$$reward = \begin{cases} v^{\text{test}}, & \text{if } 0 \le v^{\text{test}} < 1. \\ a \times v^{\text{test}} + b \times v^{\text{llm}}, & \text{if } v^{\text{test}} = 1. \end{cases}$$
(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 here is scalar, which is used to calculate the Q-value at each node and compute the score during selection. For 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 factual details about the code are crucial for making modifications but cannot be captured in a scalar reward. Therefore, in addition to the scalar reward, we incorporate verbal feedback.

Verbal Feedback When the generated code fails to pass a public test case, human programmers typically analyze the issue by examining information such as variable values during execution. In solving the code generation task using search algorithms, relying solely on scalar feedback based on the pass rate of public test cases lacks detailed information. Therefore, we introduce verbal feedback in the MCTS process. Specifically, as described in Section 3.2, we obtain block-level analysis information when the code fails a public test case and store this information as verbal feedback in the current node. This feedback is used during 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. In addition to using scalar feedback to update the value of parent nodes, verbal feedback is stored in the current leaf node for use in subsequent expansion and rethink phases.

Rethink When the code encounters a public test case that it fails to pass, we can obtain the block-level analysis as the detailed execution verbal feedback. How can we use such fine-grained feedback from the code environment to generate correct code? We propose to use this feedback to make the LLM "rethink", meaning to regenerate the current thought and thus avoid generating the previously incorrect code. As shown in Figure 1, the leaf node is re-generated by $z^{\text{new}} \sim p(z|s, f, z^{\text{old}})$. It is important to note that we do not regenerate the parent nodes of the leaf node for the following reasons: 1) The parent nodes have already undergone multiple rounds of reward collection from all their child nodes, and regenerating renders the previously collected rewards unusable. 2) The parent node has passed its own "rethink" process. That means, during the parent node's generation, the parent node either did not encounter any failing public test cases during the evaluation step or has already been refined through the "rethink" process. Therefore, we do not go back to the parent nodes again when "rethink" on the leaf node.

The advantage of introducing *rethink* is in two perspectives: from the perspective of code generation, this rethink process refines the thought process leading to the correct code. From the MCTS perspective, this rethink essentially refines current action. Since the MCTS tree is built step by step, improving the quality of the current action allows the LLM to explore more optimal paths in the nearly infinite search space, thereby enhancing the overall search quality of the tree. Through the *rethink* operation, we effectively integrate refining the code's thought process with the MCTS search process.

5 EXPERIMENT SETTINGS

Datasets We evaluate RethinkMCTS and the baseline methods on two popular benchmark datasets: APPS (Hendrycks et al., 2021) and HumanEval (Chen et al., 2021). The APPS dataset contains three levels of difficulties: introductory, interview, and competition, with a total of 5000 train problems and 5000 test problems. Since RethinkMCTS and baselines are tuning-free, we evaluate all the methods on the test set and select the formal 100 problems of each difficulty to do the evaluation. The APPS dataset does not specify public and private test cases. So, we split a program's test cases evenly into two sets. The first set is used as the public test cases for the algorithms to optimize the pass rate, and the second set is used as the private test cases for evaluating the generated programs. We use pass rate and pass@1 as the evaluation metric for code correctness following Zhang et al. (2023), where pass rate is the average percentage of the private test cases that the generated programs pass over all the problems and pass@1 is the percentage of the problems where the generated programs pass all the private test cases, which is the most widely adopted metric in the literature of code generation (Austin et al., 2021; Chen et al., 2021; Dong et al., 2023).

		Pass Rate (%)			Pass@1 (%)			
		APPS Intro.	APPS Inter.	APPS comp.	APPS Intro.	APPS Inter.	APPS comp.	HumanEval
GPT3.5-turbo	Base	50.43	40.57	23.67	29	19	9	70.12
	PG-TD	60.89	50.80	26.50	40	25	8	76.22
	ToT	63.21	63.49	26.30	37	33	11	84.15
	LATS	54.06	45.86	21.83	36	20	7	79.88
	RAP	43.22	43.32	22.83	21	14	8	71.95
	LDB	56.68	46.78	21.00	35	22	8	81.09
	Reflexion	53.20	45.58	17.50	35	21	7	71.95
	RethinkMCTS	67.09	68.65	29.50	45	38	13	89.02
GPT-4o-mini	Base	56.56	52.40	35.00	35	29	16	87.20
	PG-TD	65.87	70.37	43.16	45	46	27	91.46
	ToT	74.34	71.83	42.50	55	47	27	93.29
	LATS	69.46	67.65	35.83	50	45	19	93.29
	RAP	64.24	57.25	37.67	39	32	20	87.20
	LDB	60.64	60.78	40.33	40	38	23	90.85
	Reflexion	60.65	56.87	38.00	40	31	18	90.85
	RethinkMCTS	76.60	74.35	42.50	59	49	28	94.51

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 is 16.

Baselines To illustrate the effectiveness of RethinkMCTS, we compare with various code generation methods that are based on LLMs, including methods that use code execution feedback to refine codes: LDB (Zhong et al., 2024), Reflexion (Shinn et al., 2024), and we compare with search-based methods: PG-TD (Zhang et al., 2023), ToT (Yao et al., 2024), LATS (Zhou et al., 2023) and RAP (Hao et al., 2023).

Implementation We pick GPT-3.5-turbo and GPT-4o-mini as the backbone models to compare different algorithms. For search-based 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 equation 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 equation 3 are set to be (0.8, 0.2). 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). For ToT, we set the search action as thought and prompt each node to generate one full code based on the node's thoughts, similar to in RethinkMCTS, while it does not contain detailed feedback and the rethinking process.

6 RESULTS

Overall Performance We present the overall performance in Table 1. It can be seen that RethinkMCTS outperforms all the baselines in both datasets. Besides, by comparing with the original base model, feedback-enhanced and tree search-enhanced methods improve the performance effectively, demonstrating the effectiveness of exploring different strategies and using the detailed feedback of the coding environment. Also, we can observe that the ToT baseline achieved impressive performance, which is benefit from searching for the thought-level action space, demonstrating the advantage of searching for the reasoning process during the coding process instead of searching at token-level (PG-TD) or searching at the code-level (LATS). However, since ToT does not utilize feedback during the search process, its search quality is inferior to our approach, which refines thoughts using feedback.

Ablation Study We conduct ablation studies to remove each of our model's components and reevaluate the performance with GPT-3.5-turbo backbone. For the verbal feedback part, we compare with two variants. One is to remove the whole verbal feedback and retain only the scalar reward (w/o VF). The other one is that we change the verbal feedback to the standard error information or the wrong output of the code while removing the block-level analysis information (w/o blockInfo). The results are shown in Figure 2. The chart shows that each module contributes to the model's overall performance. Verbal feedback has the most significant impact, as the *rethink* operation we proposed is primarily based on feedback from code execution. Without this feedback, providing instructions for *rethink* alone would not be sufficient for the model to effectively refine thoughts. This demonstrates that in the context of code generation, the fundamental source for refining erro-

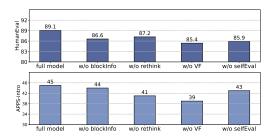


Figure 2: Ablation study of block-level analysis (blockInfo), rethink operation, the verbal feedback (VF) and self-evaluation.

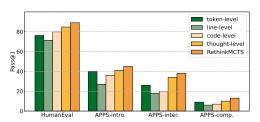


Figure 3: Performance comparison between different search granularity. For advanced model like GPT-3.5-turbo, it's better to explore directly in the thought space.

neous thoughts lies in the detailed feedback provided by the code environment. In fact, previous studies (Huang et al., 2023b) have noted that LLMs lack the ability to self-correct their reasoning without external feedback.

Additionally, we can see that for the HumanEval dataset, block-level analysis information significantly affects performance (89.1 \rightarrow 86.6), while for the APPS dataset, the impact is smaller. We hypothesize that this is due to the fewer public test cases in HumanEval compared to APPS, making fine-grained analysis of each test case crucial for *rethink* to refine erroneous thought in HumanEval. In contrast, the ample number of public test cases in APPS allows the model to find the issues with only standard error information. This is also why dual evaluation is crucial for HumanEval, as the limited number of public test cases is insufficient to fully assess the quality of a code snippet. In such cases, it becomes necessary to introduce LLM to reevaluate the code, allowing the search process to filter for better solutions. Finally, the *rethink* operation we proposed significantly enhances the results. This improvement stems from *rethink* enabling the use of fine-grained execution feedback and block-level code analysis in verbal feedback, effectively correcting logical errors in the code.

Search Granularity Study We propose to conduct a thought-level search during code generation. Here, we compare the action spaces for MCTS, specifically examining different levels of search actions: token, line, code, and thought. The experimental results are presented in Figure 3.

As shown in the figure, the thought-level search is more effective in finding viable code compared to token, line, and code-level searches. This demonstrates that for advanced LLMs like GPT-3.5-turbo, exploring the reasoning process is more important. In contrast, token-level and line-level searching may be better suited for smaller, less capable models like GPT-2 (Zhang et al., 2023). Additionally, we observe that token-level searching performs better than line and code-level searching. This is due to that with 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, the effectiveness of is further enhanced in RethinkMCTS by introducing detailed feedback and *rethink* operations, making the search over thoughts in the code generation process even more effective.

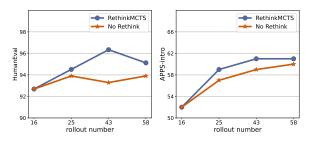


Figure 4: Performance comparison between *rethink* more times and more rollouts without *rethink*. *rethink* is more effective than increasing rollouts due to refining the wrong thoughts improve the whole search quality.

Effectiveness of *Rethink* The goal of *rethink* is to improve the quality of the thought search by refining error thoughts, thereby enabling the generation of correct code 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. And we keep the rollout numbers the same of them. The results are shown in Figure 4.

The figure shows that both increasing the number of *rethink* operations and increas-

ing the number of rollouts both enhance the performance of code generation. This is expected, as more extensive exploration raises the probability of finding the correct code. However, RethinkM-

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

Table 2: The performance comparison between using Direct Self-evaluation and Self-generating test evaluation.

CTS outperforms the version without *rethink* under the same number of explorations. From the perspective of tree search, without the *rethink* operation, incorrect thoughts are likely to persist in the parent nodes, causing the reasoning process of the expanded child nodes to continue along the wrong path. This makes it difficult to ensure the quality of the entire trace. The *rethink* operation corrects these flawed thoughts, allowing subsequent exploration to proceed more accurately, thereby improving the overall quality of the tree search.

Method	APPS-intro.	HumanEval
w/o Rethink	10.04	48.30
RETHINKMCTS	15.60	53.29

Table 3: The success rate comparison of the searched codes between with and without the rethinking operation.

Furthermore, we compare the pass rate on public test cases of all the generated codes for the entire MCTS tree, with and without the *rethink* operation. The results are presented in Table 3. We can see that the *rethink* operation increases the proportion of effective code found in the entire tree. This demonstrates that refining erroneous thoughts allows the tree to explore more along the correct paths.

Self-evaluation vs. Self-generating Unit Tests Due to the insufficient coverage of public test cases, we propose a dual evaluation approach. When the code passes all the public test cases, we supplement it with a comprehensive self-evaluation of the code. In this section, we compare self-evaluation with self-generating unit tests. In the latter approach, when the code passes the public test cases, we have the LLM generate additional test cases. The combined results serve as a comprehensive evaluation of the code. Experimental results are shown in Table 2.

The table shows that self-generating unit tests improve the *pass rate* on test cases but do not enhance the *pass@1* metric. This is because self-evaluation directly assesses the code after it passes the public test cases, scoring based on how well the code meets the problem requirements. Therefore, it provides a more accurate evaluation of the code's ability to solve the entire problem. On the other hand, self-generating unit tests are an additional, separate task that focuses more on the tests than the code. There could be two reasons why it results in a higher *pass rate* but a lower *pass@1*: 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 improve the *pass rate* by filtering code that fits the test patterns more closely. However, since it does not directly target the problem, it may only enhance the fit for specific test cases without solving more fundamental issues. 2) The generated tests may not always be correct, which can mislead the code's modification and search direction when the tests are incorrect.

7 Conclusion

In this work, we propose RethinkMCTS, the first framework that tries to search and refine the thoughts for code generation. Compared to previous tree search-based methods, RethinkMCTS explores different coding strategies by searching through the reasoning process and introduces a feedback and refining mechanism to improve the search quality. This approach effectively utilizes execution information from code generation to construct verbal feedback, thereby refining erroneous reasoning steps. In the evaluation phase, we also introduce a dual evaluation method to address the incomplete coverage of public test cases. Through comparative experiments on the APPS and Humaneval datasets, we demonstrate that RethinkMCTS achieves the highest pass rate, proving its ability to effectively use feedback to correct reasoning errors and more efficiently search for high-quality code.

REFERENCES

- V Aho Alfred, S Lam Monica, and D Ullman Jeffrey. *Compilers principles, techniques & tools.* pearson Education, 2007.
- Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. arXiv preprint arXiv:2108.07732, 2021.
- Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1):1–43, 2012.
- 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. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.
- Yihong Dong, Jiazheng Ding, Xue Jiang, Ge Li, Zhuo Li, and Zhi Jin. Codescore: Evaluating code generation by learning code execution. *arXiv* preprint arXiv:2301.09043, 2023.
- Xidong Feng, Ziyu Wan, Muning Wen, Ying Wen, Weinan Zhang, and Jun Wang. Alphazero-like tree-search can guide large language model decoding and training. *arXiv* preprint arXiv:2309.17179, 2023.
- Data Flow. Control Flow Analysis. PhD thesis, QUEENSLAND UNIVERSITY OF TECHNOL-OGY, 1994.
- Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999*, 2022.
- Shibo Hao, Yi Gu, Haodi Ma, Joshua Jiahua Hong, Zhen Wang, Daisy Zhe Wang, and Zhiting Hu. Reasoning with language model is planning with world model. *arXiv preprint arXiv:2305.14992*, 2023.
- Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, et al. Measuring coding challenge competence with apps. *arXiv preprint arXiv:2105.09938*, 2021.
- Zhiyuan Hu, Chumin Liu, Xidong Feng, Yilun Zhao, See-Kiong Ng, Anh Tuan Luu, Junxian He, Pang Wei Koh, and Bryan Hooi. Uncertainty of thoughts: Uncertainty-aware planning enhances information seeking in large language models. *arXiv preprint arXiv:2402.03271*, 2024.
- Dong Huang, Qingwen Bu, Jie M Zhang, Michael Luck, and Heming Cui. Agentcoder: Multi-agent-based code generation with iterative testing and optimisation. *arXiv preprint arXiv:2312.13010*, 2023a.
- Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Wei Yu, Xinying Song, and Denny Zhou. Large language models cannot self-correct reasoning yet. *arXiv preprint arXiv:2310.01798*, 2023b.
- James R Larus. Whole program paths. ACM SIGPLAN Notices, 34(5):259–269, 1999.
- Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*, 2023.
- Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. Wizardcoder: Empowering code large language models with evol-instruct. *arXiv preprint arXiv:2306.08568*, 2023.
- Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. Self-refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems*, 36, 2024.

- Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. Code llama: Open foundation models for code. arXiv preprint arXiv:2308.12950, 2023.
- Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36, 2024.
- David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.
- David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*, 2017.
- Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. Advances in neural information processing systems, 35:24824–24837, 2022.
- Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. *Advances in Neural Information Processing Systems*, 36, 2024.
- Shun Zhang, Zhenfang Chen, Yikang Shen, Mingyu Ding, Joshua B Tenenbaum, and Chuang Gan. Planning with large language models for code generation. *arXiv preprint arXiv:2303.05510*, 2023.
- Li Zhong, Zilong Wang, and Jingbo Shang. Ldb: A large language model debugger via verifying runtime execution step-by-step. *arXiv preprint arXiv:2402.16906*, 2024.
- Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. Language agent tree search unifies reasoning acting and planning in language models. *arXiv* preprint arXiv:2310.04406, 2023.