A Reward-Guided Dual-Phase Framework for Adaptive Inference-Time Reasoning

Anonymous ACL submission

Abstract

Large Language Models (LLMs) have made strong progress in reasoning. To enhance the reasoning performance, a common inference-time approach is tree-based search, which decomposes the reasoning process into multiple steps, expands multiple reasoning paths, and uses reward models to prune and select candidates. However, based on our exploration, the simple decomposition may lead to suboptimal searching efficiency: while planning is generally harder, it is the execution errors that are more likely to propagate to later steps. This indicates that planning and execution play different roles in reasoning and should be treated differently during tree-based search. Given this, to enhance the searching efficiency, we propose a dual-phase test-time scaling framework that separates reasoning into planning and execution, and performs search over each phase independently. To further refine the algorithm, we also introduce a dynamic budget allocation mechanism that adaptively redistributes sampling effort based on reward feedback, allowing early stopping on confident steps and reallocation of computation to more challenging steps. Experiments on both math reasoning and code generation benchmarks demonstrate that our approach consistently improves accuracy while reducing redundant computation.

1 Introduction

In recent years, Large Language Models (LLMs) have achieved remarkable success in complex reasoning tasks such as mathematical problem solving, code generation, and decision making (Chen et al., 2021; Yao et al., 2023). A common approach to improve LLM reasoning is Chain-of-Thought (CoT) prompting (Wei et al., 2022), which guides the model to generate intermediate reasoning steps in a stepwise manner, often improving performance on arithmetic and symbolic tasks.

To further enhance the quality and accuracy of multi-step reasoning, recent works have explored test-time scaling methods, which perform structured search or sampling over multiple reasoning paths. According to Snell et al. (2024), these methods consider two main directions: (1) **Distribution-Based Sampling**, by refining how candidates are generated, e.g., through iterative revision (Muennighoff et al., 2025; Shinn et al., 2023) or parallel sampling with selection (Snell et al., 2024; Diao et al., 2023); and (2) **Reward-Based Searching**, by using verifiers or reward models to select or guide promising reasoning paths (Snell et al., 2024; Wu et al., 2024). Among the latter, Process Reward Modeling (PRM) (Wu et al., 2024) has shown strong performance by evaluating partial reasoning steps and guiding the search process accordingly. Instead of judging only final outcomes, PRM provides supervision at the process level, assigning rewards to intermediate steps and enabling the model to distinguish between useful and unproductive reasoning trajectories when generating intermediate steps. This step-level feedback allows search algorithms to prune low-quality candidates earlier and concentrate computation on promising directions.

While existing PRM-based test-time scaling methods improved the reasoning performance of LLMs, several key limitations remain. First, existing literature on test-time scaling methods scales up the computation based on simple decomposition of the whole reasoning process, and there is limited understanding on how test-time scaling behaves beyond the simple decomposition. In particular, reasoning for complex tasks such as mathematical problem solving and code generation inherently exhibits a hierarchical structure consisting of two distinct cognitive phases: planning, which entails high-level strategic formulation (e.g., "define variables and identify the next subgoal"), and execution, which precedes low-level derivation or implementations (e.g., arithmetic calculations or code

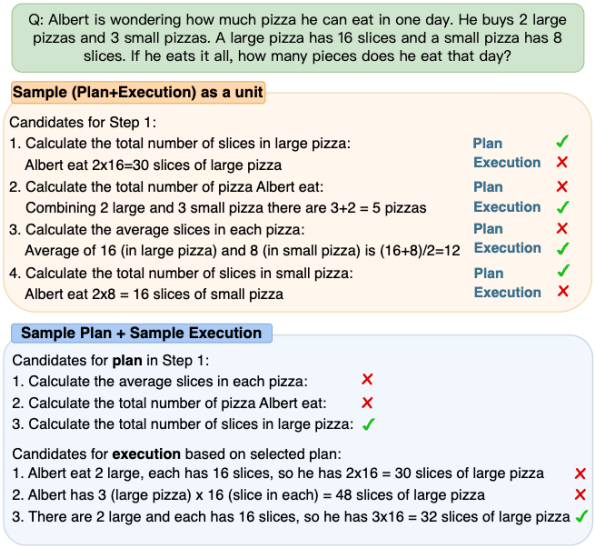

Figure 1: An example of reasoning with plan and execution as a single unit versus searched separately.

writing) (Zhou et al., 2022; Wang et al., 2024b; Hao et al., 2023; Wang et al., 2023a). Although the benefits of explicitly writing out plans during reasoning have been discussed (Zhou et al., 2022; Wang et al., 2024b; Hao et al., 2023; Wang et al., 2023a), most literature in test-time scaling treats planning and execution as a unified pipeline: the model generates a plan immediately followed by its execution, and both are evaluated together.

However, since planning and execution behave very differently, such a coupled strategy is suboptimal. As discussed in Section 4.1, planning is typically more uncertain, making it more likely to be incorrect, whereas execution is usually simpler but its mistakes are more fatal, as they tend to propagate through subsequent steps. When these two phases are evaluated as a single unit, their mismatched behaviors lead to systematic inefficiencies. For example, as shown in Figure 1, for a math problem, if a step is already flawed at the planning stage (Candidate 2 in sampling Plan+Execution as a unit), the search still wastes budget generating its executions even if the execution calculates correctly based on the planning. Conversely, if the planning is correct but the execution occurs an error in the calculation, then the subsequent steps will use the incorrect number for later steps (Candidate 1).

The second limitation of existing PRM-based test-time scaling methods is that they mainly use a fixed sampling budget for each step within each example (e.g., sampling k candidates at each step), ignoring the varying difficulty across different steps, examples and datasets. This rigid allocation can lead to inefficient computation, especially when simple steps receive excessive attention while more challenging parts remain underexplored. While there are some studies that explore sample-wise budget allocation, i.e., dynamically distributing the overall budget across different questions or different candidate trajectories (Zuo and Zhu, 2025; Lin et al., 2025), these approaches do not address the problem of deciding the budget for each individual step within a single reasoning trajectory.

To address these limitations, we propose *DREAM, a Dual-phase REward-guided Adaptive reasoning framework at test tiMe*. Unlike prior methods that treat each plan–execution pair as a single unit, DREAM explicitly conducts search in two stages: it first searches over multiple planning candidates and uses a reward model to select promising subgoals; then, conditioned on the selected plans, it searches over execution candidates and applies a second reward evaluation to retain the most reliable solutions. For example, for the question in Figure 1, we first sample candidate plans and select the promising ones. Then, conditioned on these plans, we generate multiple execution candidates. This two-stage procedure ensures that poor plans are eliminated early, while promising plans can be paired with different execution attempts until the correct result is found. In addition, we further incorporate DREAM with a *dynamic budget allocation* mechanism that adaptively adjusts the number of samples at both phases based on real-time reward feedback, enabling early stopping on easy steps and reallocating resources to harder ones.

To validate the effectiveness of the proposed algorithm, we conduct comprehensive evaluations across two domains: math reasoning and code generation. Experimental results show that our approach not only improves answer accuracy but also enhances test-time efficiency.

2 Related works

Test-time Scaling. Test-time scaling method improves reasoning without parameter updates by expending more computation at inference. According to Snell et al. (2024), two main mechanisms for test-time scaling include (1) Distribution-Based Sampling and (2) Reward-Based Searching. Methods of (1) include s1 (Muennighoff et al., 2025) and Reflexion (Shinn et al., 2023), which introduces sequential self-revision to iteratively refine candidate solutions, and Best-of-N (Wang et al., 2022), which samples multiple candidate reasoning chains in parallel and aggregates by majority vote.

Methods of (2) work by treating intermediate reasoning states as nodes in a search tree and expand continuations via the base LLM. They include MCTS-based methods, such as RAP (Hao et al., 2023), LiteSearch (Wang et al., 2024a), rStar (Qi et al., 2024) and rStar-Math (Guan et al., 2025), which apply Monte Carlo Tree Search to explore reasoning paths, and verifier-based methods, which rely on outcome-level judges (Cobbe et al., 2021; Snell et al., 2024) or process-supervised reward models (PRMs) (Lightman et al., 2023; Wu et al., 2024; Hooper et al., 2025) to score and prune candidates. Moreover, Setlur et al. (2025) indicate that verifier-based methods combined with search-based strategy are provably better than verifier-free approaches. While effective, most current methods (even those that adopt a plan-execution format) still treat planning and execution as a single unified process, without performing separate search or adaptive budget allocation across the two phases. Moreover, although a variety of test-time methods have been proposed, in our experiments we mainly consider reward model-based methods as baselines to ensure a fair comparison, as reward models provide additional information beyond the base LLM.

More related works about Code Generation with LLMs can be found in Appendix C.

3 Preliminary Explorations

In this section, we present preliminary explorations examining whether plan and execution exhibit fundamentally different behaviors and error patterns.

Settings. We numerically examine the behaviors of the planning and execution steps in GSM8K. Since current LLMs do not explicitly separate the planning and execution steps, we use few-shot prompts to induce the model to generate reasoning in a separated plan–execution format to make this structure explicit and observable. As shown in the "Sample (Plan+Execution) in a unit" in Figure 1, the prompt expresses the *plan* as a sub-question or subgoal, and the *execution* as a direct response that completes that subgoal through concrete derivations. The whole few-shot prompt is shown in Appendix K.

Observations. Given the above decomposition approach, we evaluate plan and execution steps from different perspectives and make comparisons:

First, we use the perplexity to evaluate the confidence of the LLM towards the planning and the execution step. As in Jelinek et al. (1977), perplexity is related to the probability of the generated sequence given the original input, thus can be used to reflect the confidence of the generation. The results are summarized in Table 1. From the table, it is clear that the perplexity of the planning and execution are different, demonstrating that the LLM is generally less confident in the planning step.

Table 1: Distribution of Perplexity and Reward

		mean	median	std	min	max
Per.	plan	5.4646	3.8493	6.456	1.0679	88.4054
	exec	1.2614	1.2297	0.1831	1.0024	2.3179
Rew.	plan	0.804	0.957	0.274	0.008	1.000
	exec	0.768	0.957	0.319	0.002	1.000

Second, we also check the distribution of the reward value of the planning and execution steps. (See Section 4.3 for details on the reward model.) The results are also in Table 1. Based on the results, the reward for the execution steps is indeed lower than that for the planning steps. To explain this, we manually checked some of the samples and have the following observation: if the planning makes a mistake, the later planning steps can still search and possibly reach the correct result. However, for the execution steps, once a step gives a wrong number, it is very likely that the later steps will use this number for further calculation. Some illustrative examples supporting these observations are provided in Appendix H.

Implications. The above two observations together indicate that both planning and execution are important and they should be treated differently throughout the searching in test-time scaling (the proposed dual-phase search algorithm in Section 4.1): Since the planning steps are in general harder, we need to use a dedicated reward model for these steps and allocate essential computation budgets. On the other hand, since the consequence of mistakes in the execution steps is more severe, we should also use another reward model for this specific purpose and allocate enough budgets to these steps. While both steps need computation budgets, since different datasets have their specific levels of difficulties in planning and execution steps, it is essential to dynamically assign the budgets (the proposed dynamic budget allocation strategy in Section 4.2), and there will be a **synergistic effect** when using the dual approach and the dynamic budget allocation together (Figure 4 in Section 5.2).

4 Method

In this section, we present the details of the proposed dual-phase search method and the dynamic budget allocation strategy. Since the concrete im-

Table 2: Distribution of the reward values before and after reward-based selection

		mean	median	std	min	max
before	plan	0.804	0.957	0.274	0.008	1.000
	exec	0.768	0.957	0.319	0.002	1.000
after	plan	0.863	0.973	0.216	0.033	1.000
	exec	0.834	0.977	0.269	0.007	1.000

plementation of test-time scaling differs across task types, following prior work (Hao et al., 2023; Li et al., 2024), we use two representative tasks (math reasoning and code generation) as examples to illustrate our method. For clarity, this section mainly focuses on the math reasoning setting, while the algorithm for code generation is shown in Appendix A.

4.1 Dual-phase Search over Plan/Execution

Our dual-phase search builds on the standard beam search framework, whose workflow is shown in Figure 2 (a). In standard beam search, we utilize the standard reasoning format, where plan and execution are not explicitly expressed, and use the reasoning model to sample a fixed number of candidates. Reward models are applied to score all candidates, and the top-ranked ones are retained for expansion in the next step. Given the preliminary explorations in Section 3, we further extend beam search into a variant that outputs plan-execution pairs in a single step, as illustrated in Figure 2(b).

Guided by the earlier evidence that plan and execution behave differently, we further decouple the search over the two roles. Our dual-phase search (Figure 2(c)), explicitly separates each step into a plan phase (red nodes) and an execution phase (blue nodes). In the plan phase, N_1 candidate subgoals are sampled and scored by a plan reward model (PRM_{plan}), and the top n_1 candidates are selected. In the execution phase, N_2 continuations are generated conditional on the chosen plans and scored by an execution reward model (PRM_{exec}). Then the top n_2 candidates are selected for expansion in the next step. This separation allows weaker plans to be pruned early, while promising plans can receive multiple execution attempts, reducing the risk of discarding good strategies due to execution errors.

To quickly check the effectiveness of the proposed idea, we provide further empirical evidence that separating the search over plan and execution is beneficial. Specifically, we report the distributions of plan and execution rewards before any reward-based selection is applied, as well as the distributions observed at subsequent steps after selection. The results are shown in Table 2. By comparing the two stages, we find that the reward values for both plan and execution increase noticeably in the later steps, demonstrating that the separated selection process effectively filters out weaker structural decisions and derivations as reasoning progresses.

4.2 Adaptive Budget Allocation

Motivation for Adaptive Budget Allocation While dual-phase search improves search efficiency, the method in Figure 2(c) still assigns an equal budget to plan and execution at every step. In practice, this uniform allocation is not optimal. The reason is threefold. First, the difficulty of plan and execution is not the same, and even within a single problem, their difficulty can vary from step to step. Second, difficulty also varies across datasets and across individual examples within the same dataset. For instance, using the same model (Qwen2.5-1.5B-Math), performance is roughly 82% on GSM8K but only about 60% on MATH. Moreover, within GSM8K itself, some problems can be solved in 1 step, whereas others require up to 8 steps. Third, different LLMs implicitly separate planning and execution in a different manner, leading to different computational needs for each phase. These reasons suggest that allocating the same sampling budget to every step of every example is inefficient: easy steps waste resources, while difficult steps remain underexplored. This motivates an adaptive budget allocation mechanism, whose details will be introduced in the following paragraph.

Implementation of Adaptive Budget Allocation. We further extend our method to a *budget-adjusted dual-phase search* that incorporates an adaptive allocation strategy that stops early when confident candidates are already found and reallocates additional computation to more challenging steps, improving the overall accuracy–efficiency trade-off. The specific workflow is shown in Figure 2 (d) and the detailed algorithm can be found in Algorithm 1 in Appendix D.

At each step, sampling in both the plan and execution phases follows a two-threshold rule. Specifically, as candidates are sampled and scored, if at least n_1 (for planning) or n_2 (for execution) candidates exceed a specific threshold τ_{p1} (for planning) or τ_{e1} (for execution), sampling is terminated early without consuming the full budget. Conversely, if after exhausting the full budget, there is no candidate whose reward value higher than a lower threshold (τ_{p2}/τ_{e2}),[1] we allow at most an additional m_1

[1] In practice, for simplicity, we set τ_{p1}=τ_{e1} and τ_{p2}=τ_{e2}

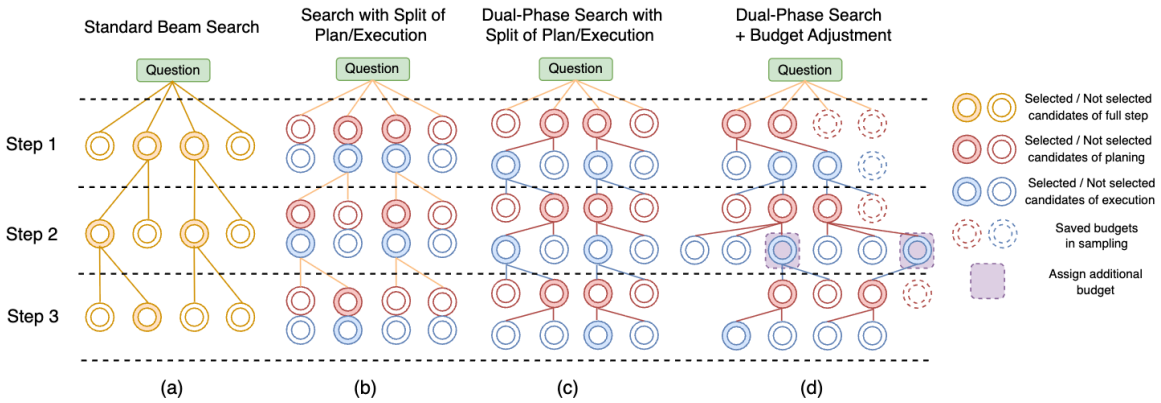

Figure 2: Workflow of Standard Beam Search and Dual-Phase Search (with budget allocation).

(planning) or m_2 (execution) samples to be generated to search harder steps more thoroughly.

This mechanism prevents computation waste on easy steps with confident high-reward candidates, while allocating extra exploration to uncertain or challenging steps. By combining dual-phase scoring with this adaptive budget policy, our method aligns computation with step-level difficulty and improves efficiency over standard beam search (as will be demonstrated in in Section 5.2). Furthermore, as discussed in Section 5.2, the combination of dual-phase search and dynamic budget allocation has a synergy effect: the two components mutually reinforce each other by reducing wasted computation and reallocating resources to harder steps.

Remark. We note that dual-phase search is not limited to math or code reasoning, but can generalize to a wide range of reasoning tasks, as long as the solutions can be expressed in a plan–execution format and the reasoning can be organized in a tree-based framework. To illustrate this, in Appendix G, we present additional examples beyond math and code tasks showing how other tasks can be naturally decomposed into plan and execution.

4.3 Construction of the Reward Models

Training data. To develop the reward models, we follow Wang et al. (2023b) to construct datasets that evaluate the quality of both planning and execution at each reasoning step. We first generate complete multi-step reasoning trajectories. For each question in the training set, we sample multiple trajectories at a higher decoding temperature to ensure diversity. Each trajectory is expressed as a sequence of step-wise plan–execution pairs that progressively lead to the final solution. To annotate the steps, we adopt a rollout-based labeling strategy. For each intermediate plan or execution, we generate five independent continuations beginning from that step using the same LLM that produced the trajectory. If at least one rollout leads to a correct final answer, the current step is labeled as "+"; otherwise –". This method assesses the utility of a plan/execution by its downstream impact on solving the problem.

Reward function. We obtain the reward model by fine-tuning an instruction-tuned LLM. The input to the reward model, denoted as x, consists of the original question, all preceding reasoning steps (including plans and executions), and the current plan/execution to be evaluated. For the output, instead of adding a separate classification head, we follow Dong et al. (2024) to reformulate the prediction as a next-token prediction task: the final position of the input sequence is reserved for a binary label, and the model is trained to output either "+" or "–" at that position. The reward function is:

$$\text{Reward}(x) = \frac{\exp(\ell_+(x))}{\exp(\ell_+(x)) + \exp(\ell_-(x))}$$

where $\ell_+(x)/\ell_-(x)$ are the logits output by the model when predicting the special tokens "+"/"–".

5 Experiments

We evaluate our method on both math reasoning and code-generation tasks. In this Section, we mainly present and discuss the math reasoning results, while the corresponding experiments for code generation are provided in Appendix B.

5.1 Setup

We evaluate our method on two widely used math reasoning benchmarks: GSM8K (Cobbe et al., 2021) and MATH (Hendrycks et al., 2021). For GSM8K, we use the full training set of around 7.5k problems to construct reward-model training data and evaluate on the 1.3k test set. For MATH, we use the 12.5k training problems to build reward-model data and conduct evaluation on the standard MATH500 benchmark. We generate large-scale synthetic trajectories to build up the training

dataset: about 400k samples for GSM8K and 400k samples for MATH. The training trajectories and the rollout process for label assignment are produced using LLaMA-3-8B-Instruct (for GSM8K) and Qwen-2.5-3B-Instruct (for MATH). We then combine data from both datasets to train the reward models, fine-tuning Qwen-2.5-32B-Instruct (Yang et al., 2024b). The resulting reward model is applied in experiments on both benchmarks.

We compare our method with three baselines: majority vote, standard beam search, and REBASE (Wu et al., 2024), a tree-based search method that does not implement dual-phase search. For a fair comparison, we format all reasoning in the plan–execution style and use the same reward model (trained with the rollout-based annotation strategy) across all methods that rely on reward signals. For evaluation, we use three LLMs on each benchmark: Qwen-2.5-MATH-1.5B-Instruct (Yang et al., 2024a), DeepSeekMath-7B-Instruct (Shao et al., 2024), and LLaMA-3-8B-Instruct for GSM8K / LLaMA-3.1-8B-Instruct (Grattafiori and et al., 2024) for MATH. We use different LLaMA variants for the GSM8K and MATH experiments to ensure that the reasoning models have moderate ability relative to the difficulty of each dataset. This choice allows us to better demonstrate the effectiveness of test-time scaling, since improvements are more evident when the base model is neither too strong nor too weak. More details about the training and inference configurations and the settings of budgets and thresholds are shown in Appendix F.

5.2 Main Results.

In this subsection, we present the main experimental results for math reasoning tasks to demonstrate the effectiveness of DREAM in achieving a better accuracy-efficiency trade-off in reasoning. We show the accuracy–tokens frontier of each method in Figure 3 and one additional comparison in Figure 4. For DREAM, we consider the variants with/without budget allocation, which are labeled “DREAM" and “DREAM(+)", respectively.

Based on the results shown in Figure 3 and Figure 4, we have the following observations.

First, all tree-based search methods consistently achieve a significantly better accuracy–efficiency trade-off than majority vote. For example, on the MATH dataset with LLaMA-3.1-8B-Instruct, the performance gap can reach up to 20%. This confirms the effectiveness of tree-structured search mechanisms in improving the accuracy–efficiency trade-off. In addition, the consistently strong performance across models also highlights the effectiveness of the reward model trained with the rollout-based annotation strategy.

Second, DREAM outperforms standard beam search which does not explicitly separate planning and execution in searching. For example, on the MATH dataset with Qwen2.5-MATH-1.5B, DREAM continues to outperform beam search, and the advantage becomes more pronounced as the token budget increases. This suggests sampling and selecting planning and execution independently provides a more effective search of reasoning steps and leads to higher-quality candidate trajectories.

Third, DREAM(+) with dynamic budget allocation provides additional gains in accuracy–efficiency trade-off. For instance, on GSM8K with LLaMA-3-8B-Instruct, DREAM(+) consistently achieves about a 2% improvement in accuracy over DREAM at comparable token budgets. While in some cases (often on the MATH dataset), where the problems are more challenging, the improvement over standard dual-phase search is marginal, this can be explained by the fact that the adaptive budget mechanism is more effective when step difficulty is highly variable. When every step in a trajectory is uniformly hard, reallocating budget provides little benefit, and performance is mainly constrained by the inherent capacity of the reasoning model and the reward model: In GSM8K, a large percentage (around 80%) of reasoning steps trigger early stopping, whereas in MATH this percentage is much smaller (around 5%), i.e., problems in MATH are uniformly hard.

Fourth, there is synergistic effect between DREAM and the dynamic budget allocation. As mentioned in Section 3, since the planning and execution steps face different difficulties, dynamically assigning budgets into the steps may result in a better performance. We compare the benefits of applying budget allocation to dual-phase search versus standard beam search. Figure 4 plots the accuracy gain brought by adaptive budget allocation under the same token budget. The x-axis shows the accuracy of the base method (DREAM/Beam Search), and the y-axis shows the improvement obtained by adding adaptive allocation. The figure shows that, although budget allocation consistently improves both methods, the improvement is consistently larger for DREAM(+), even at higher base accuracies where gains typically diminish. These results highlight a synergistic effect: adaptive bud-

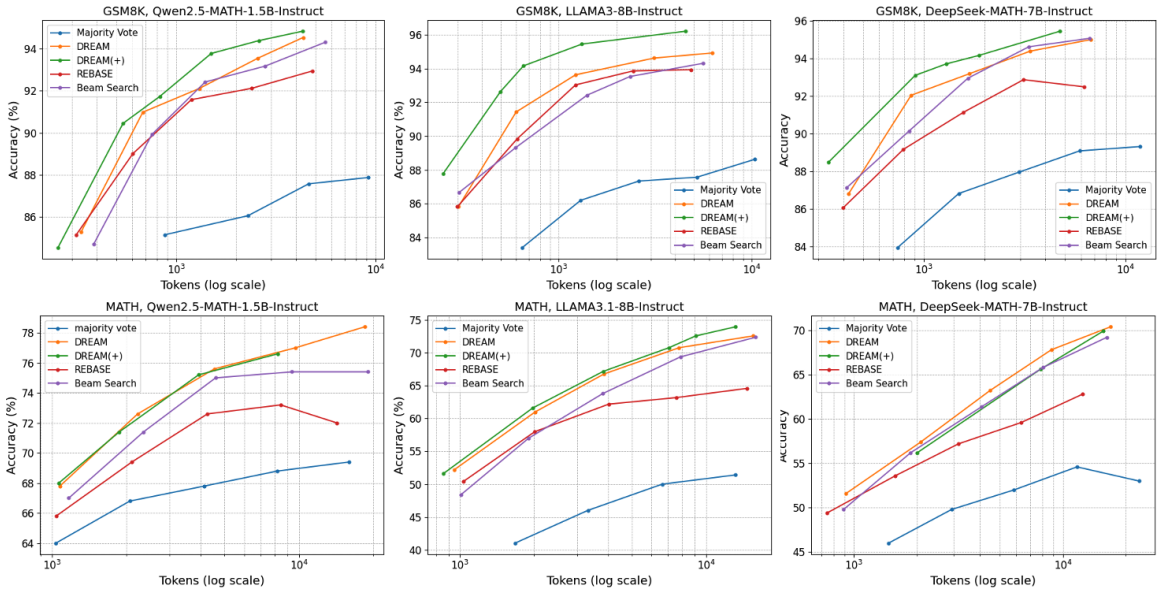

Figure 3: Accuracy vs Tokens (log) on GSM8K/MATH datasets

Table 3: Performance in out-of-distribution datasets

AMC23 (LLaMA3.1)						ADSIV (LLaMA3)					
DREAM(+)		DREAM		Majority Vote		DREAM(+)		DREAM		Majority Vote	
acc	# tokens	acc	# tokens	acc	# tokens	acc	# tokens	acc	# tokens	acc	# tokens
37.50%	3186.38	30.00%	3149.6	22.50%	2851.45	95.35%	160.55	93.02%	162.65	93.02%	193.93
47.50%	6068.18	42.50%	6256.20	22.59%	5469.45	96.35%	218.76	95.35%	332.60	94.02%	388.29
50.00%	11210.10	47.50%	12430.3	27.50%	11708.9	97.67%	543.02	96.01%	650.10	95.35%	782.80
60.00%	23515.20	50.00%	22971.6	30.00%	22586.3	98.01%	1112.1	97.34%	1314.7	95.02%	1558.1

get allocation is more effective when combined with DREAM's dual-phase search.

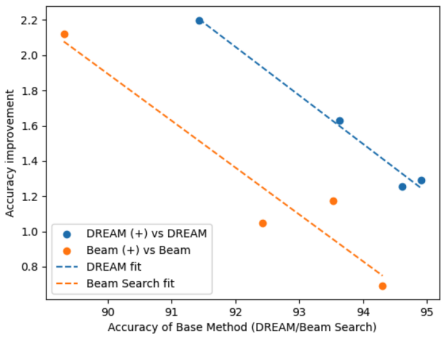

Figure 4: Accuracy-Tokens Frontier across methods.

Finally, the reward model can generalize well across different settings. We note that in many of our settings, the models used to develop the training data for the reward model are different from the models used in the final evaluation. The only in-distribution setting is LLaMA-3-8B-Instruct with GSM8K, while all other evaluation settings are out of distribution. Nevertheless, in the out-of-distribution settings, DREAM consistently demonstrates strong performance, indicating that our reward model generalizes well across different backbone LLMs instead of overfitting to the one used in training. This strong transferability stems from how the reward model's training data is obtained. Because the data exposes diverse reasoning behaviors and failure modes, the reward model learns model-agnostic indicators of reasoning quality (e.g., logical consistency and step-wise correctness), rather than surface patterns tied to a particular model.

Remark on FLOPs. While the above observations are obtained based on Figure 3 where number of tokens measures the computation cost, the observations are the same when using FLOPs (FLOP-gen) and considering the additional computation involved in the reward-model evaluation (FLOP-rew). Table 5 shows the results for Llama3-8B-Instruct on GSM8K, with all methods evaluated under a tree-search width of 16. We also include an additional configuration of DREAM+ using a smaller search width 8 (labeled as DREAM(8)+) to approximately match the accuracy of baselines. The results show that FLOP-gen increase approximately linearly with the number of generated tokens. Besides, the total FLOP-rew remains less than 1/10 of FLOP-gen in the implementation of DREAM since reward scoring only requires a single forward pass over the partial sequence without any autoregressive decoding. These finding indicate that the trend of the overall FLOPs is approximately the same as the number of tokens, further verifying the

Table 4: Performance comparison across different reward model size.

GSM8K						MATH					
Qwen2.5-32B		Qwen2.5-7B		Majority vote		Qwen2.5-32B		Qwen2.5-7B		Majority vote	
acc	# tokens	acc	# tokens	acc	# tokens	acc	# tokens	acc	# tokens	acc	# tokens
91.43%	601.13	87.34%	606.191	83.40%	646.18	61.00%	2021.16	55.20%	1829.73	41.00%	1674.98
93.63%	1221.21	93.63%	1191.49	86.20%	1297.45	66.80%	3854.31	58.40%	3576.18	46.00%	3314.41
94.62%	2463.82	94.62%	2385.57	87.34%	2598.09	70.80%	7776.58	62.80%	7477.92	50.00%	6647.12
94.92%	4916.89	94.92%	4444.57	87.57%	5202.77	72.60%	15553.2	65.60%	14557.8	51.40%	13157.1

Table 5: Computation of each method on GSM8K

	DREAM	BEAM	REBASE	MAJ	DREAM(8)+
acc	94.62%	93.53%	93.86%	87.34%	**93.86%**
tokens	2442.21	2339.38	2426.49	2598.10	**1508.04**
FLOP-gen	4.34e+15	4.17e+15	4.31e+15	4.63e+15	**2.68e+15**
FLOP-rew	3.43e+14	1.77e+14	1.84e+14	0	**1.14e+14**
FLOP-sum	4.68e+15	4.35e+15	4.49e+15	4.63e+15	**2.79e+15**

computation efficiency of our approaches.

5.3 Generalization of the Reward Model to Unseen Datasets

In this section, we examine the reward model's transferability in the math reasoning domain. Specifically, we consider two out-of-distribution datasets: AMC23 (Math-AI, 2023), which includes 40 competition-style problems, and the test set of ASDiv (Miao et al., 2021), which contains 301 grade-school math word problems. The experiments are conducted with DREAM/DREAM(+) using LLaMA3/3.1-8B-Instruct, and compared with majority vote. The results are presented in Table 3.

From Table 3, we observe that across both datasets, DREAM with our reward model consistently achieves higher accuracy at the same level of tokens compared to majority vote, and DREAM(+) further improves the accuracy–efficiency trade-off. Notably, on AMC23, which is significantly more challenging than the datasets used to train the reward model, our approach still provides strong guidance for reasoning steps, showing up to a 30% improvement over majority vote. This suggest that our reward model generalizes beyond its training distribution and demonstrates strong transferability to out-of-distribution math reasoning tasks.

5.4 Ablation Studies

In this section, we study the reward model size for math reasoning (Section 5.4.1), and the empirical and theoretical analysis on the selection of the thresholds. Due to page limit, the threshold studies are postponed to Appendix I.2.

5.4.1 Reward Model Size

In this subsection, we study the impact of reward-model size. Our main math-reasoning experiments use a reward model fine-tuned from Qwen2.5-32B-Instruct. To assess whether smaller models can serve as effective alternatives, we additionally fine-tune a reward model based on Qwen2.5-7B-Instruct. We then compare the performance of standard dual-phase search (without budget allocation) under two configurations: using Qwen2.5-7B-Instruct or Qwen2.5-32B-Instruct as the reward model. For reference, we also report results with majority vote. We conduct the experiments using LLaMA3/3.1-8B-Instruct as the reasoning models and the results are summarized in Table 4.

According to the results, the 32B reward model consistently outperforms the 7B version, achieving better accuracy-efficiency trade-offs across datasets. This suggests that, as larger instruction-tuned LLMs possess stronger intrinsic reasoning and representation capabilities, fine-tuning them with rollout-based supervision enables the reward function to better capture nuanced signals of correctness. Nevertheless, the 7B reward model also demonstrates strong effectiveness: although its step selection is not as precise as the 32B model, it still achieves significantly better accuracy–efficiency trade-offs than majority vote. This suggests that, with properly labeled training data, even moderately sized reward models can provide substantial benefits while reducing computations.

6 Conclusion

In this work, we proposed DREAM, a dual-phase test-time scaling framework that explicitly separates reasoning into plan and execution and performs dedicated search in each phase. By equipping both phases with reward models and introducing adaptive budget allocation, DREAM provides finer-grained control over search, reduces wasted computation, and improves accuracy on math reasoning and code generation. Empirically, DREAM consistently outperforms standard beam search and prior PRM-based methods, demonstrating the benefits of separating plan and execution and adaptively managing computation.

Limitations

Although our approach improves test-time reasoning through structured search, it relies on relatively large reward and reasoning models. Increasing model size typically provides stronger guidance and higher-quality search, but it also comes with additional computation and memory costs.

References

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*.

Sahil Chaudhary. 2023. Code alpaca: An instruction-following llama model for code generation. https://github.com/sahil280114/codealpaca.

Jingchang Chen, Hongxuan Tang, Zheng Chu, Qianglong Chen, Zekun Wang, Ming Liu, and Bing Qin. 2024. Divide-and-conquer meets consensus: Unleashing the power of functions in code generation. *Advances in Neural Information Processing Systems*, 37:67061–67105.

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*.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, and 1 others. 2021. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*.

Shizhe Diao, Pengcheng Wang, Yong Lin, Rui Pan, Xiang Liu, and Tong Zhang. 2023. Active prompting with chain-of-thought for large language models. *arXiv preprint arXiv:2302.12246*.

Hanze Dong, Wei Xiong, Bo Pang, Haoxiang Wang, Han Zhao, Yingbo Zhou, Nan Jiang, Doyen Sahoo, Caiming Xiong, and Tong Zhang. 2024. Rlhf workflow: From reward modeling to online rlhf. *arXiv preprint arXiv:2405.07863*.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. 2023. Pal: Program-aided language models. In *International Conference on Machine Learning*, pages 10764–10799. PMLR.

Aaron Grattafiori and et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*.

Xinyu Guan, Li Lyna Zhang, Yifei Liu, Ning Shang, Youran Sun, Yi Zhu, Fan Yang, and Mao Yang. 2025. rstar-math: Small llms can master math reasoning with self-evolved deep thinking. *arXiv preprint arXiv:2501.04519*.

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

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. 2021. Measuring mathematical problem solving with the math dataset. *arXiv preprint arXiv:2103.03874*.

Coleman Hooper, Sehoon Kim, Suhong Moon, Kerem Dilmen, Monishwaran Maheswaran, Nicholas Lee, Michael W Mahoney, Sophia Shao, Kurt Keutzer, and Amir Gholami. 2025. Ets: Efficient tree search for inference-time scaling. *arXiv preprint arXiv:2502.13575*.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, and 1 others. 2024. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*.

Fred Jelinek, Robert L Mercer, Lalit R Bahl, and James K Baker. 1977. Perplexity—a measure of the difficulty of speech recognition tasks. *The Journal of the Acoustical Society of America*, 62(S1):S63–S63.

Dacheng Li, Shiyi Cao, Chengkun Cao, Xiuyu Li, Shangyin Tan, Kurt Keutzer, Jiarong Xing, Joseph E Gonzalez, and Ion Stoica. 2025. S*: Test time scaling for code generation. *arXiv preprint arXiv:2502.14382*.

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

Hunter Lightman, Vineet Kosaraju, Yuri Burda, Harrison Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. 2023. Let's verify step by step. In *The Twelfth International Conference on Learning Representations*.

Junhong Lin, Xinyue Zeng, Jie Zhu, Song Wang, Julian Shun, Jun Wu, and Dawei Zhou. 2025. Plan and budget: Effective and efficient test-time scaling on large language model reasoning. *arXiv preprint arXiv:2505.16122*.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is your code generated by chatGPT really correct? rigorous evaluation of large language models for code generation. In *Thirty-seventh Conference on Neural Information Processing Systems*.

Math-AI. 2023. Amc23: Competition-level math problems. https://huggingface.co/datasets/math-ai/amc23. Hugging Face Dataset, accessed 2025-09-17.

Shen-Yun Miao, Chao-Chun Liang, and Keh-Yih Su. 2021. A diverse corpus for evaluating and developing english math word problem solvers. *arXiv preprint arXiv:2106.15772*.

Niklas Muennighoff, Zitong Yang, Weijia Shi, Xiang Lisa Li, Li Fei-Fei, Hannaneh Hajishirzi, Luke Zettlemoyer, Percy Liang, Emmanuel Candès, and Tatsunori Hashimoto. 2025. s1: Simple test-time scaling. *arXiv preprint arXiv:2501.19393*.

Ziyi Ni, Yifan Li, Ning Yang, Dou Shen, Pin Lv, and Daxiang Dong. 2024. Tree-of-code: A tree-structured exploring framework for end-to-end code generation and execution in complex task handling. *arXiv preprint arXiv:2412.15305*.

Zhenting Qi, Mingyuan Ma, Jiahang Xu, Li Lyna Zhang, Fan Yang, and Mao Yang. 2024. Mutual reasoning makes smaller llms stronger problem-solvers. *arXiv preprint arXiv:2408.06195*.

Amrith Setlur, Nived Rajaraman, Sergey Levine, and Aviral Kumar. 2025. Scaling test-time compute without verification or rl is suboptimal. *arXiv preprint arXiv:2502.12118*.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Yang Wu, and 1 others. 2024. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36:8634–8652.

Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. 2024. Scaling llm test-time compute optimally can be more effective than scaling model parameters. *arXiv preprint arXiv:2408.03314*.

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*.

Lei Wang, Wanyu Xu, Yihuai Lan, Zhiqiang Hu, Yunshi Lan, Roy Ka-Wei Lee, and Ee-Peng Lim. 2023a. Plan-and-solve prompting: Improving zero-shot chain-of-thought reasoning by large language models. *arXiv preprint arXiv:2305.04091*.

Peiyi Wang, Lei Li, Zhihong Shao, RX Xu, Damai Dai, Yifei Li, Deli Chen, Yu Wu, and Zhifang Sui. 2023b. Math-shepherd: Verify and reinforce llms step-by-step without human annotations. *arXiv preprint arXiv:2312.08935*.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2022. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*.

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.

Yangzhen Wu, Zhiqing Sun, Shanda Li, Sean Welleck, and Yiming Yang. 2024. Inference scaling laws: An empirical analysis of compute-optimal inference for problem-solving with language models. *arXiv preprint arXiv:2408.00724*.

An Yang, Beichen Zhang, Binyuan Hui, Bofei Gao, Bowen Yu, Chengpeng Li, Dayiheng Liu, Jianhong Tu, Jingren Zhou, Junyang Lin, and 1 others. 2024a. Qwen2. 5-math technical report: Toward mathematical expert model via self-improvement. *arXiv preprint arXiv:2409.12122*.

An Yang and 1 others. 2024b. Qwen2.5: Technical report. *arXiv preprint arXiv:2412.15115*.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. React: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*.

Zhaojian Yu, Yinghao Wu, Yilun Zhao, Arman Cohan, and Xiao-Ping Zhang. 2025. Z1: Efficient test-time scaling with code. *arXiv preprint arXiv:2504.00810*.

Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Claire Cui, Olivier Bousquet, Quoc Le, and 1 others. 2022. Least-to-most prompting enables complex reasoning in large language models. *arXiv preprint arXiv:2205.10625*.

Bowen Zuo and Yinglun Zhu. 2025. Strategic scaling of test-time compute: A bandit learning approach. *arXiv preprint arXiv:2506.12721*.

A Dual-Phase Search for Code Generation — Algorithm and Implementation Details

For code generation tasks, we follow the framework of CodeTree (Li et al., 2024) and extend it with our dual-phase search. We adopt this framework because its tree-based structure provides a natural backbone for implementing our dual-phase design. The key difference from math reasoning is that, in code generation task, a visible test set is available for debugging. Instead of treating each step as a partial components of the solution, in CodeTree, each step produces a complete program, and subsequent steps perform iterative debugging based on execution results from failed test cases of earlier solutions.

In the original CodeTree algorithm, both the initial generation and subsequent debugging involve sampling multiple planning candidates per step, which are then attempted sequentially. Whether to expand the current step or backtrack to alternative candidates depends on whether the current node's reward exceeds that of the previous one. The node's reward value is computed by combining two factors: (i) the percentage of passed test cases, and (ii) a score given by an LLM-based critic agent.

We modify the above framework in several ways to implement the dual-phase search. First, for each step, we apply a dedicated reward model to the N_1 sampled planning candidates, rank them, and prioritize higher-scoring plans for execution. Second, in the execution phase, we scale generation by producing N_2 candidate solutions conditioned on the chosen plan, and select the one with the highest reward. Third, as evidenced by experimental results shown in Appendix I.1, the critic agent provided only marginal benefit. As a result, we remove this component and rely solely on the percentage of passed test cases as the execution reward. Finally, we incorporate a budget-adjustment criterion similar to that in mathematical reasoning: if the reward of a generated candidate exceeds a threshold, we stop further sampling and save the unused budget; if the rewards of all sampled candidates fall below another threshold, additional budget is allocated to generate more candidates. Through these modifications, we extend CodeTree into a dual-phase search framework that conducts separate searches for planning and execution, integrates reward methods for each phase, and allocates computation adaptively based on step-level difficulty.

B Experiment Settings and Results for Code Generation Task

B.1 Experiment Setting

For code generation, we conduct experiments on HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021), including their extended versions (HumanEval+ and MBPP+) (Liu et al., 2023), which include more challenging test cases. The reward model training data is drawn from the MBPP training set (approximately 600 examples), augmented with around 3,000 examples from CodeAlpaca (Chaudhary, 2023). The generation of training trajectories and the labeling process are carried out using a group of Qwen2.5-Coder models ranging in size from 1.5B to 32B. We use models with different capacities to produce a wider range of trajectories (both correct and incorrect), which improves the diversity of supervision for training the reward model. The final reward model is obtained by fine-tuning Qwen-2.5-Coder-7B-Instruct (Hui et al., 2024). We compare our method against the standard CodeTree (Li et al., 2024) and Reflexion (Shinn et al., 2023), which prompts the LLM to repeatedly reflect on its previously generated code based on test case results. The evaluations are conducted with two LLMs including LLaMA-3.1-8B-Instruct, Qwen-2.5-Coder-1.5B-Instruct (Hui et al., 2024).

B.2 Main Results

In this subsection, we present the main results of DREAM combined with CodeTree on code generation benchmarks, comparing with Reflexion and the standard CodeTree method. Figure 5 shows the accuracy–token curve of each approach. Consistent with the math reasoning experiments in Section 5.2, we report two variants of DREAM: the version with/without budget allocation (denoted as "DREAM/DREAM(+)").

There are several observations from the experiment. **First,** according to the results in Figure 5, we observe that across all settings and datasets, the performance of CodeTree is significantly improved when combined with dual-phase search using reward models. For example, when the computation budget is around 10^3 tokens, CodeTree+DREAM achieves an accuracy about 10% higher than CodeTree on both MBPP and HumanEval with Qwen2.5-Coder-1.5B-Instruct. This demonstrates the effectiveness of separated searching of planning and execution and leveraging dedicated reward sig-

nals for each phase. In addition, applying budget allocation consistently provides further gains in the accuracy–efficiency trade-off. This suggests that adaptively allocating computation not only improves accuracy but also reduces unnecessary generation cost. **Second,** in some cases, when the token budget is small, Reflexion also achieves a strong accuracy–efficiency trade-off. For instance, with LLaMA-3.1-8B-Instruct, Reflexion performs better when the token budget (log scale) is below 10^3. However, as the budget increases, the accuracy of Reflexion grows more slowly compared to CodeTree+DREAM, indicating that Reflexion saturates earlier, while dual-phase search continues to benefit from additional computation. **Finally**, it is also worth noting that HumanEval does not appear in the training data of the reward model. The advantage of CodeTree+DREAM in Figure 5 also demonstrates the strong generalization ability of our reward model to unseen datasets.

In short, these findings highlight the advantages of integrating dual-phase search and adaptive budget allocation into tree-based code generation framework.

C Related Works about Code Generation with LLMs

Recent work has explored diverse strategies (including test-time scaling) to enhance LLMs for code generation. For example, S*(Li et al., 2025) employs parallel sampling with sequential scaling and adaptive input synthesis to improve code generation. (Yu et al., 2025) introduce Z1, which trains the LLM on both short and long reasoning trajectories and leverages a shifted thinking window to enable the model to adaptively control the length of its 'thinking' process according to problem complexity. In addition, PlanSearch (Wang et al., 2024b) boosts code generation by exploring diverse natural-language plans before translating them into code. In addition, tree-structured or agent-based searching frameworks like CodeTree (Li et al., 2024), Tree-of-Code (Ni et al., 2024) and Funcoder (Chen et al., 2024) design stepwise generation or refinement algorithms for code generation, where candidate programs are expanded or revised through structured search, demonstrating the benefits of structured exploration.

Although many of the above code generation methods also introduce a planning stage before the generation of code, they typically focus only on improving or selecting a good plan to guide execution rather than performing separate search processes for planning and execution. In contrast, our work performs dual-phase scaling and selection, and assigns budget across phases, which has the potential of having a more effective use of computation.

D Algorithm of DREAM(+)

We present the detailed algorithm of DREAM(+) in Algorithm 1.

E Examples to demonstrate difficulty variants in reasoning steps.

In this subsection, we present real examples to illustrate difficulty diversity both across problems in a dataset and across steps within a single problem, motivating the need for dynamic allocation. Tables 6 and 7 provide examples from GSM8K and MATH: for each dataset, we include one easy problem, where intermediate steps have high average reward values and low variance, and one hard problem, where intermediate steps have lower average reward values and higher variance. For reference, we also provide the ground-truth solution for each example, offering a more intuitive sense of the difficulty of each problem. These examples demonstrate that difficulty variance arises not only across problems but also within individual reasoning trajectories.

F Additional Details of Experiments Setting

In this subsection, we provide additional details of experiment settings to ensure reproducibility.

F.1 Training and Inference Configurations

- For math reasoning, we develop the reward model by fine-tuning Qwen2.5-32B-Instruct for 2 epochs with a learning rate of 2.0×10^{-6}, using the Adam optimizer and a cosine learning-rate scheduler. The fine-tuning is conducted on 8 H100 GPUs with a batch size of 32. For code generation, we fine-tune Qwen2.5-Coder-7B-Instruct for 3 epochs, while keeping the other hyperparameters the same.

- During inference on math reasoning, we set the sampling temperature to 1.0 for LLaMA3/3.1-8B-Instruct and DeepSeek-MATH-7B-Instruct, and 0.5 for

Algorithm 1 Dual-Phase Search with budgets adjustment.

Require: Question Q, Planning/Execution Budget N_1/N_2, Thresholds τ_{p_1}/ τ_{p_2} and τ_{e_1} /τ_{e_2}, Beam Width n_1/n_2, Additional Budget limit m_1/m_2.

Initialize finished paths $F \leftarrow \emptyset$
Initialize step counter $s \leftarrow 1$
Initialize beam set $B \leftarrow \emptyset$
while $s \leq$ max_steps **do**
  if all $b \in B$ is finished **then**
    break
  end if
  C_p(candidates storage) $\leftarrow \emptyset$
  if $s = 1$ **then**
    Generate up to N_1 candidates for planning p_1 with reward scores r
    Early stop if there are n_1 planning all having reward $r > \tau_{p_1}$
    $C_p \leftarrow C_p \cup \{(p_1^{(i)}, r_1^{(i)}) \mid i = 1, \ldots, n\}$, n denotes the number of actual sampling
    if all $r < \tau_{p2}$ **then**
      Generate up to additional m_1 candidates for planning p_1 with reward scores r
      Early stop if there are n_1 planning having $r > \tau_{p1}$
      $C_p \leftarrow C_p \cup \{(p_1^{(i)}, r_1^{(i)}) \mid i = 1, \ldots, n\}$
    end if
  else
    for all $b \in B$ **do**
      Generate up to N_1/n_1 candidates for planning p_s with reward scores r
      Early stop if there are N_1/n_1 planning having $r > \tau_{p1}$
      $C_p \leftarrow C_p \cup \{(p_s^{(i)}, r_s^{(i)}) \mid i = 1, \ldots, n\}$
      if all $r < \tau_{p2}$ **then**
        Generate up to m_1 additional candidates for planning p_s with reward scores r
        Early stop if there are N_1/n_1 planning having $r > \tau_{p1}$
        $C_p \leftarrow C_p \cup \{(p_s^{(i)}, r_s^{(i)}) \mid i = 1, \ldots, n\}$
      end if
    end for
  end if
  Sort C_p and take the top-n_1 planning $\{(p_1^1, e_1^1, \cdots, p_{s-1}^1, e_{s-1}^1, p_s^1, r_s^1), \cdots, (p_1^{n_1}, e_1^{n_1}, \cdots, p_s^{n_1}, r_s^{n_1})\}$
  Update beam $B \leftarrow \{(Q, p_1^1, e_1^1, \cdots, p_{s-1}^1, e_{s-1}^1, p_s^1), \cdots, (Q, p_1^{n_1}, e_1^{n_1}, \cdots, p_{s-1}^{n_1}, e_{s-1}^1, p_s^{n_1})\}$
  C_e(candidates storage) $\leftarrow \emptyset$
  for all $b \in B$ **do**
    Generate up to N_2/n_2 candidates for executions e_s based on p_s with reward scores r'
    Early stop if there are N_2/n_2 executions having reward $r' > \tau_{e1}$
    $C_e \leftarrow C_e \cup \{(p_s^{(i)}, e_s^{(i)}, {r'}_s^{(i)}) \mid i = 1, \ldots, n\}$, n denotes the number of actual sampling
    if all $r' < \tau_2$ **then**
      Generate up to m_2 additional candidates for executions e_s with reward scores r'
      Early stop if there are N_2/n_2 executions having reward $r' > \tau_{e1}$
      $C_e \leftarrow C_e \cup \{(p_s^{(i)}, e_s^{(i)}, {r'}_s^{(i)}) \mid i = 1, \ldots, n\}$
    end if
  end for
  Sort C_e and take the top-n_2 executions $\{(p_1^1, e_1^1, \cdots, p_s^1, e_s^1, {r'}_s^1), \cdots, (p_1^{n_2}, e_1^{n_2}, \cdots, p_s^{n_2}, e_s^{n_2}, {r'}_s^{n_2})\}$
  Update beam $B \leftarrow \{(Q, p_1^1, e_1^1, \cdots, p_s^1, e_s^1), \cdots, (Q, p_1^{n_2}, e_1^{n_2}, \cdots, p_s^{n_2}, e_s^{n_2})\}$
  $s \leftarrow s + 1$
end while
return Path with highest reward in B

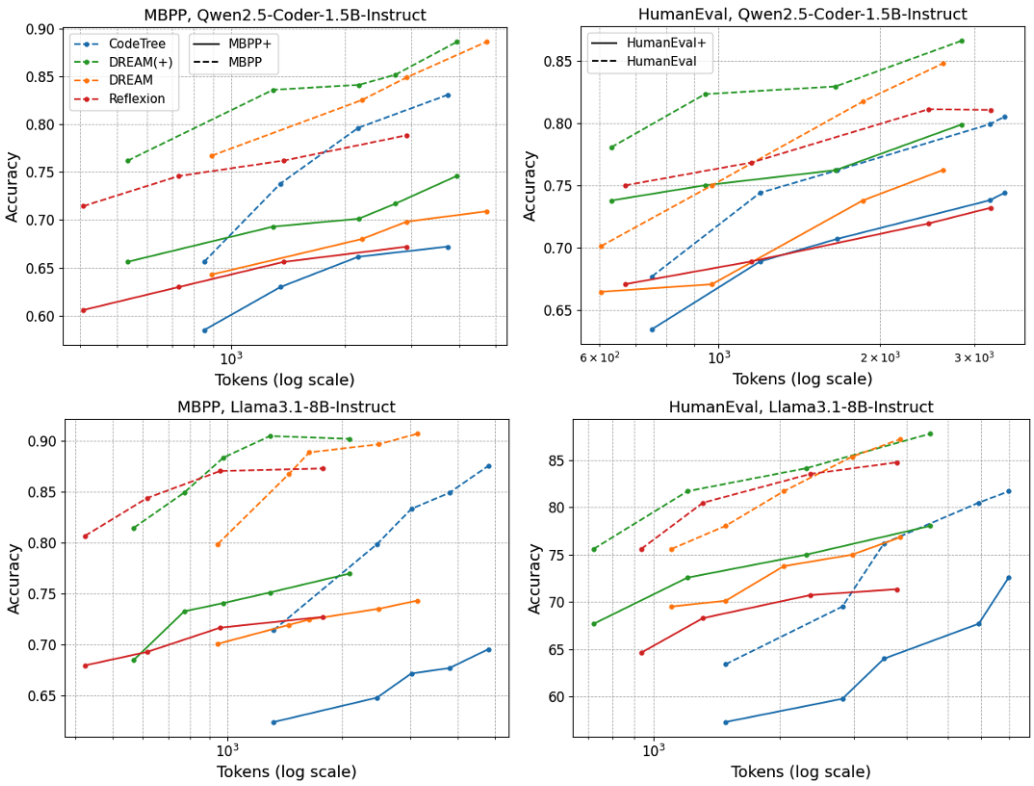

Figure 5: Accuracy vs Tokens (log) on MBPP/HumanEval datasets

Table 6: Examples of samples with different diversity in GSM8K datasets

Q: Josh decides to try flipping a house. He buys a house for $80,000 and then puts in $50,000 in repairs. This increased the value of the house by 150%. How much profit did he make? Ground truth solution: The cost of the house and repairs came out to $80,000 + 50,000 = 130,000$ He increased the value of the house by $80,000 * 1.5 = 120,000$ So the new value of the house is $120,000 + 80,000 = 200,000$ So he made a profit of $200,000 - 130,000 = 70000$ #### 70000 Average reward score for intermediate steps: 0.3613 Standard deviation of reward score: 0.184845
Q: A robe takes 2 bolts of blue fiber and half that much white fiber. How many bolts in total does it take? Ground truth solution: It takes $2/2 = 1$ bolt of white fiber So the total amount of fabric is $2 + 1 = 3$ bolts of fabric #### 3 Average reward score for intermediate steps: 0.9990 Standard deviation of reward score: 0.001746

Qwen2.5-MATH-1.5B-Instruct. In each setting in the experiments, each example is evaluated once following the corresponding test-time scaling protocol.

- For code generation, we set the temperature to 1.0 across all experiments in DREAM and DREAM(+), while using 0.0 for CodeTree and Reflexion.

Table 7: Examples of samples with different diversity in MATH datasets

Q: What is the smallest positive integer n such that all the roots of $z^4 + z^2 + 1 = 0$ are n^{th} roots of unity?

Ground Truth:
Multiplying the equation $z^4 + z^2 + 1 = 0$ by $z^2 - 1 = (z-1)(z+1)$, we get $z^6 - 1 = 0$.
Therefore, every root of $z^4 + z^2 + 1 = 0$ is a sixth root of unity.
The sixth roots of unity are e^0, $e^{2\pi i/6}$, $e^{4\pi i/6}$, $e^{6\pi i/6}$, $e^{8\pi i/6}$, and $e^{10\pi i/6}$.
We see that $e^0 = 1$ and $e^{6\pi i/6} = e^{\pi i} = -1$, so the roots of $z^4 + z^2 + 1 = 0$
are the remaining sixth roots of unity, namely $e^{2\pi i/6}$, $e^{4\pi i/6}$, $e^{8\pi i/6}$, and $e^{10\pi i/6}$.
The complex number $e^{2\pi i/6}$ is a primitive sixth root of unity, so by definition,
the smallest positive integer n such that $(e^{2\pi i/6})^n = 1$ is 6.
Therefore, the smallest possible value of n is $\boxed{6}$.

Average reward score for intermediate steps: 0.4206503
Standard deviation of the reward score: 0.3246094

Q: Simplify $\sqrt{242}$.

Ground truth:
Factor 242 as $11^2 \cdot 2$.
Then $\sqrt{242} = \sqrt{11^2} \cdot \sqrt{2} = \boxed{11\sqrt{2}}$.

Average reward score for intermediate steps: 0.9947
Standard deviation of reward score: 0.0024

Table 8: Candidate value of beam widths and sampling budgets

N_1	2	4	8	16	32
N_2	2	4	8	16	32
n_1	1	2	2	4	4
n_2	1	2	2	4	4

F.2 Settings of threshold and budgets.

For GSM8K, we use $\tau_{p1} = \tau_{e1} = 0.99$, $\tau_{p2} = \tau_{e2} = 0.9$, and m_1=m_2=2 for all models. For MATH, we set $\tau_{p1} = \tau_{e1} = 0.9$, $\tau_{p2} = \tau_{e2} = 0.5$, and m_1=m_2=2. To control test-time scaling, we vary the sampling budget N_1, N_2 and the beam widths n_1, n_2. The candidate values used in our experiments are shown in Table 8; each column represents one configuration, corresponding to a single point in the accuracy–budget curves in the main results in Figure 3. Increasing these values proportionally increases inference-time computation.

Notably, we set $N_1 = N_2$ and $n_1 = n_2$ to keep the search space balanced between the planning and execution phases. Since both phases play important roles in producing a high-quality reasoning step, using symmetric widths prevents the search from being unintentionally biased toward one phase and also avoids introducing additional hyperparameter tuning.

G Examples beyond Math and Code Tasks

To emphasize that the hierarchical plan–execution structure is not restricted to a narrow class of tasks, we note that many real-world reasoning problems inherently follow a plan–execution pattern. Beyond the math and code-generation tasks discussed in this paper, we provide additional examples below showing that other common tasks can also be naturally decomposed into planning and execution.

1. Multi-hop QA Task
Question:
In what year was the creator of the current arrangement of the 'Simpson's Theme' born?
Solution:
Step 1: Plan: Identify who created the current arrangement of the Simpson's Theme.
Execution: The current arrangement of the Simpson's Theme was created by Alf Clausen.

Step 2: Plan: Find the birth year of this person. Execution: Alf Clausen was born in 1941. So the answer is 1941.

2. Commonsense QA Task
Question: To locate a choker not located in a jewelry box or boutique where would you go?
A: jewelry store; B: neck; C: jewelry box; D: jewelry box; E: boutique
Solution:
Step 1: Plan: Understand what a "choker" is and where it normally is when being worn. Execution: A choker is a type of necklace that is typically worn around a person's neck.
Step 2: Plan: Use the condition "not located in a jewelry box or boutique" to eliminate options and find where it would be instead.
Execution: If the choker is not in a jewelry box or boutique, the natural remaining place to find it is on someone's neck, so the correct choice is B: neck.

3. Tool/Function calling tasks
Query: Find the current temperature in the capital city of France and determine whether it is higher than 10°C.
Solution:
Step 1: Plan: Identify the capital city of France using a knowledge lookup tool. This information is needed before querying the weather.
Execution: Call CountryInfoAPI (country ='France') → returns capital = "Paris".
Step 2: Plan: Using the capital city obtained in Step 1, query a weather API to get the current temperature in Paris, then compare it with 10°C.
Execution: Call WeatherAPI (city ='Paris', date ='today') → return temperature = 12°C. Since 12°C > 10°C, the answer is yes, it is higher than 10°C.

These examples demonstrate that the plan–execution structure naturally emerges in a wide variety of LLM tasks beyond the domains we evaluate in the main paper.

H Empirical Examples

In this section, we provide empirical examples to support the observation that planning errors can be recoverable through later search (Example 1), whereas execution errors tend to propagate and become harder to correct (Example 2).
Example 1:
Question: Kylar went to the store to buy glasses for his new apartment. One glass costs $5, but every second glass costs only 60% of the price. Kylar wants to buy 16 glasses. How much does he need to pay for them?
Answer:
Step 1: Calculate the discount price of one glass: 60/100 * 5 = $3.
Step 2: Calculate the average price of the glass: ($3 + $ 5)/2 =4.
Step 3: Determine the number of cheaper glasses Kylar is going to buy: Since every second glass is cheaper, Kylar will buy 16 / 2 = 8 cheaper glasses.
Step 4: Determine the amount Kylar will pay for the cheaper glasses: 8 * 3 = $24.
Step 5: Determine the amount Kylar will pay for the regular-price glasses: 8 * 5 = $40.
Step 6: Calculate the total Kylar need to pay for the glasses: 24 + 40 = $64.

Example 2:
Question: Josh decides to try flipping a house. He buys a house for $80,000 and then puts in $50,000 in repairs. This increased the value of the house by 150%. How much profit did he make?
Answer:
Step 1: Calculate sum of the cost of the house and repairs: 80,000 + 50,000 = $130,000
Step 2: Calculate how much he increase the value of the house: 80,000 * 1.5 = $140,000
Step 3: Obtain the new value of the house: 140,000 + 80,000 = $220,000
Step 4: Calculate the profit: 220,000 - 130,000 = $90,000

In Example 1, the planning in the second step provides a wrong direction; however, later steps adopt a new plan that corrects the reasoning and still arrive at the correct answer. By contrast, in Example 2, the calculation in Step 2 is incorrect, and this erroneous intermediate value is carried through all subsequent steps. Even though the high-level plan there is structurally reasonable (compute total cost, compute new value, subtract to get profit), the execution error contaminates the entire trajectory and results in an incorrect answer. Taken together, these examples illustrate an important difference: while planning mistakes can sometimes be recovered through later search, execution mistakes tend to propagate and are much more likely to be fatal.

I Additional Ablation Studies

I.1 Usage of Critic Agents in Code Generation

In this subsection, we provide empirical evidence showing that when applying dual-phase search (DREAM) in the CodeTree method, the critic agent does not yield clear benefits in performance and, in fact, reduces efficiency. Table 9 presents results for DREAM+CodeTree under settings with and without critic agents. The accuracies on MBPP/HumanEval and MBPP+/HumanEval+ are reported as "weak acc" and "acc," respectively.

From the results, we observe that to achieve the same level of reasoning accuracy, DREAM with a critic agent consistently requires substantially more generated tokens. This indicates that, although the critic agent proposed by Li et al. (2024) was originally shown to improve accuracy given sufficient computation budget, its efficiency is inferior to simply scaling generation multiple times and directly applying the percentage of passed test cases as the reward signal. These findings support our design choice of removing the critic agent from the code generation framework, simplifying the system while preserving performance and improving efficiency.

I.2 Ablation Studies on the thresholds.

In this subsection, we provide (1). Justification about why τ_{e_1}=τ_{p_1}; and (2). Theoretical and empirical analysis on how different choices of the thresholds $\tau_{e1}, \tau_{p1}, \tau_{e2}, \tau_{p2}$ affect the performance of DREAM+.

I.2.1 Justification about why τ_{e_1}=τ_{p_1}

To justify why τ_{e_1}=τ_{p_1}, empirically, as shown in Table 10, using the same threshold (τ_{e_1}=τ_{p_1}) yields slightly better performance.

The intuition behind this observation is twofold. Firstly, although planning and execution operate on different sampling spaces, their reward scores share a comparable numerical scale: both reward models evaluate step-level reasoning quality (whether a subgoal is sensible, or whether a subexecution is correct). Since the rewards are trained under the same supervision paradigm, applying the same threshold yields consistent pruning behavior in both phases.

Secondly, Different thresholds can easily create imbalanced pruning: Both planning and execution are essential for producing a good reasoning step. If one phase is pruned more aggressively than the other, the search becomes unbalanced. In such cases, either valid plans are discarded too early, or promising plans cannot be properly expanded. Therefore, using the same threshold keeps the pruning strength aligned across phases and prevents either side from dominating the search.

I.3 Theoretical and empirical analysis on the influence of the thresholds.

Empirical Results. In Table 11, we provide empirical results on how different values of $\tau_{e1}, \tau_{p1}, \tau_{e2}, \tau_{p2}$ influence the performance of DREAM+.

The left block reports results under a fixed high early step threshold ($\tau_{e1} = \tau_{p1} = 0.99$) while varying the extra-budget thresholds τ_{e2}/τ_{p2}. The right block reports results under a fixed low extra-budget thresholds ($\tau_{e2} = \tau_{p2} = 0.3$) while varying the early stop thresholds τ_{e1}/τ_{p1}.

From the results in Table 11, we observe two consistent trends.

First, when τ_1 is fixed, decreasing τ_2 leads to lower accuracy and fewer generated tokens. This result is expected, as a higher τ_2 imposes a stricter extra budget standard, causing the model to more frequently determine that the current candidates are not sufficient and therefore allocate more budget to explore further. As τ_2 decreases, fewer steps trigger budget expansion, reducing exploration and slightly lowering accuracy.

Second, when τ_2 is fixed, decreasing τ_1 also reduces accuracy and the number of generated tokens. This is because the thresholds τ_1 determine whether the remaining budget for a step should continue to be used. A higher τ_1 corresponds to a stricter stopping criterion and encourages generating more candidates within the step, thereby resulting in higher accuracy and higher computation costs.

Overall, τ_1 and τ_2 control different aspects of intra-step search, and their effects on accuracy and efficiency follow intuitive patterns.

Theoretical Analysis: Markov-based Analysis of the Threshold Behaviors.

To build a theoretical understanding of how the thresholds τ_1 and τ_2 influence DREAM+, we begin by modeling the within-step search as a simplified Markov process. To begin, we model the single step generation in DREAM+ as a simple Markov process with states $\{1, 2, 3, ..., n, *\}$ where state $*$ denotes the end state. For any states a and b, let $P(x_{i+1} = a \mid x_i = b) = P_{b \to a}$. We adopt the following assumptions for the transition matrix.

Table 9: Comparison of performance with/without critic agent

MBPP(+)						HumanEval(+)					
without critic			with critic			without critic			with critic		
weak acc	acc	# tokens	weak acc	acc	# tokens	weak acc	acc	# tokens	weak acc	acc	# tokens
81.49%	68.52%	569.59	81.49%	68.52%	1051.52	75.61%	67.69%	721.28	75.61%	67.69%	1235.92
86.51%	74.10%	771.66	87.58%	74.88%	1348.16	81.71%	72.57%	1205.10	78.66%	70.13%	1596.59
88.37%	74.09%	976.72	86.25%	73.02%	1627.35	84.14%	72.57%	1619.51	80.49%	71.34%	1827.49
90.48%	75.14%	1292.83	88.37%	76.19%	2074.12	84.14%	75.00%	2307.87	81.71%	71.95%	2311.83
90.21%	76.98%	2081.73	90.73%	75.94%	2620.38	87.80%	78.04%	4552.75	87.20%	76.83%	4833.02

Table 10: Performance of DREAM in gsm8k with Qwen2.5-MATH-1.5B-Instruct when τs take different values. (We set $N_1 = N_2 = 4$, $n_1 = n_2 = 2$)

$\tau_2 = 0.7$			$\tau_2 = 0.9$		
τ_{p_1}/τ_{e_1}	**acc**	**tokens**	τ_{p_1}/τ_{e_1}	**acc**	**tokens**
0.9/0.8	89.40%	501.276	0.99/0.9	90.00%	569.846
0.8/0.9	89.00%	492.466	0.9/0.99	91.20%	525.868
0.85/0.85	90.00%	492.182	0.95/0.95	91.60%	522.408

Table 11: Performance of DREAM+ with different values of thresholds (GSM8k, Qwen2.5-MATH-1.5B-Instruct)

$\tau_{e_1} = \tau_{p_1} = 0.99$			$\tau_{e_2} = \tau_{p_2} = 0.3$		
τ_{e_2}/τ_{p_2}	**acc**	**tokens**	τ_{e_1}/τ_{p_1}	**acc**	**tokens**
0.9/0.9	0.92	538.754	0.99/0.99	0.892	489.742
0.7/0.7	0.902	520.430	0.8/0.8	0.884	472.432
0.5/0.5	0.896	506.496	0.6/0.6	0.878	467.584
0.3/0.3	0.892	489.742	0.4/0.4	0.876	465.156

1. $P_{b\to b+1} = \frac{1}{2}$ for any $1 \le b < n$
2. $P_{a\to *} = \frac{1}{2}$ for any $1 \le a \le n$
3. $P_{n\to n} = \frac{1}{2}$

At step i, each beam samples multiple candidates for next states and selects the one with the highest reward value based on the reward model. This process can be modeled as: at state $x_i = b$, we draw several independent samples of x_{i+1} under a sampling budget k, and retain the candidate with the highest reward. We further assume that the reward assigned by the reward model is positively correlated with the transition likelihood toward the end state $*$, $P^{\infty}_{x_{i+1}\to *}$. Then we have the following theorem:

Theorem I.1 *Under the assumptions described above, suppose the maximum number of reasoning steps is M (M is sufficiently large). Then starting from state b, the probability of eventually reaching the terminal state $*$ within the remaining $M - i$ steps is: $P^{(M-i)}_{b\to *} = 1 - 2^{-(M-i-b+1)}$. Furthermore, if at step i we sample k candidates for x_{i+1} and keep the one with the highest reward, then the probability of successfully reaching the end state becomes: $P^* = 1 - (\frac{1}{2^{M-i-b+1}})^k$.*

Based on the above threshold, we have the following two propositions on the impact of τ_1 and τ_2 towards P^*. For simplicity, each proposition considers τ_1 and τ_2 respectively.

Proposition I.1 *Let $p = 1 - \frac{1}{2^{(M-i-b)}}$. Assume the sampling budget is and $\tau_2 = 0$. Then:*
(1) If $\tau_1 \in (0, p]$, the probability of successfully reaching the end states is: $P^* = 1 - (\frac{1}{2^{M-i-b+1}})$.
(2) If $\tau_1 \in (p, 1]$, $P^* = 1 - (\frac{1}{2^{M-i-b+1}})^k$
When $k > 1$, P^* is higher when $\tau_1 \in (p, 1]$.

Proposition I.2 *Let $p = 1 - \frac{1}{2^{(M-i-b)}}$. Assume that the original sampling budget is k, the maximum additional budget triggered by τ_2 is k', and $\tau_1 = 1$ is fixed. Then:*
(1) If $\tau_2 \in (0, p]$, the probability of successfully reaching the end states is: $P^* = 1 - (\frac{1}{2^{M-i-b+1}})^k$.
(2) If $\tau_2 \in (p, 1]$, $P^* = 1 - (\frac{1}{2^{M-i-b+1}})^{(k+k')}$
Thus, P^* is higher when $\tau_2 \in (p, 1]$.

The two proposition provide evidence that, when all the other conditions are fixed, increasing either τ_1 or τ_2 increase the probability of successfully reaching the end state increase. This theoretical analysis explains why larger thresholds generally lead to better performance in DREAM+, as they effectively allocate more sampling budget to promising reasoning paths.

For simplicity, our analysis considers the effect of each threshold independently by fixing one and varying the other. Intuitively, increasing both τ_1 and τ_2 at the same time would compound their effects, which tightening selection and increasing the effective sampling budget, thereby further enhancing the probability of reaching the correct solution.

J Additional Evaluation

J.1 Experiments in more challenging datasets and more capable models

To provide further empirical evidence for the effectiveness of our method, in this subsection we additionally evaluate Qwen2.5-MATH-72B (Yang et al., 2024b), one of the strongest math-reasoning models, on two harder benchmarks: AMC23 (Math-AI, 2023) and GSM8K-Hard (Gao et al., 2023). In these experiments, we use the same reward model as the one used in the experiments of Section 5.2. The results are presented in Table 14 and 13 and plot in Figure 6. From the results, we can observe that, DREAM consistently achieves a better accuracy–efficiency trade-off across both datasets compared with the baselines, demonstrating that the benefits of our method generalize to stronger models and more difficult reasoning tasks.

J.2 Comparison between plan/execution decomposition and increasing granularity of CoT

In this subsection, we provide comparison between plan/execution decomposition and increasing granularity of CoT to further demonstrate that the effectiveness of our method does not come from simply making the reasoning steps more fine-grained, but from the structural decomposition that separates high-level planning from low-level execution.

To begin with, we would like to highlight the fundamental difference between plan–execution decomposition and simply increasing step granularity. Specifically, increasing granularity means dividing a single reasoning step into smaller sequential segments (e.g., splitting one step into two micro-steps). In contrast, plan–execution decomposition separates two different types of reasoning behaviors. The planning component focuses on forming high-level strategic decisions or subgoals, while the execution component focuses on concretely carrying out these decisions. Consequently, decomposing a step into plan and execution changes the structure of the reasoning process rather than merely adjusting the level of granularity. To better illustrate the difference, we provide real examples demonstrating both strategies and the combination of both strategies:

Q: Jack deposited $4000 in a bank with a 1% annual interest rate. What will be the total amount after one year?

1. Standard CoT:
Step 1: With 1% interest rate, after the first year, the total value is 10000×(1+1%)=10100
Step 2: With 2% interest rate, after the second year, the total value is 10100(1+2%)=10302

2. CoT with increased granularity:
Step 1: With 1% interest rate, the first y ear's interest is 10000*1%=100.
Step 2: The total value after the first year is 10000+100=10100.
Step 3: With 2% interest rate, the second year's interest 10100*2% = 202.
Step 4: The total value after the second year is 10100+202=10302.

3. CoT with plan/execution decomposition:
Step 1: Determine the total value after the first year:
- Given that the interest rate of the 1st year is 1%. The total value becomes 10000×(1+1%)=10100
Step 2: Determine the total value after the second year:
- Given that the interest rate of the 1st year is 2%. The total value becomes 10100×(1+2%)=10302.

4. CoT with plan/execution decomposition + increased granularity:
Step 1: Determine the first year's interest:
- Given that the interest rate of the 1st year is 1%. The first year's interest is 1% x 10000=100
Step 2: Determine the total value after the first year:
- After the first year, the interest is 10000+100=10100
Step 3: Determine the second year's interest:
- Given that the interest rate of the 2nd year is 2%. The second year's interest is 2% x 10100=202
Step 4: Determine the total value after the second year:
- After the second year, the total value becomes 10100+202 = 10302.

To better demonstrate that the improvement of DREAM comes from structural plan–execution de-

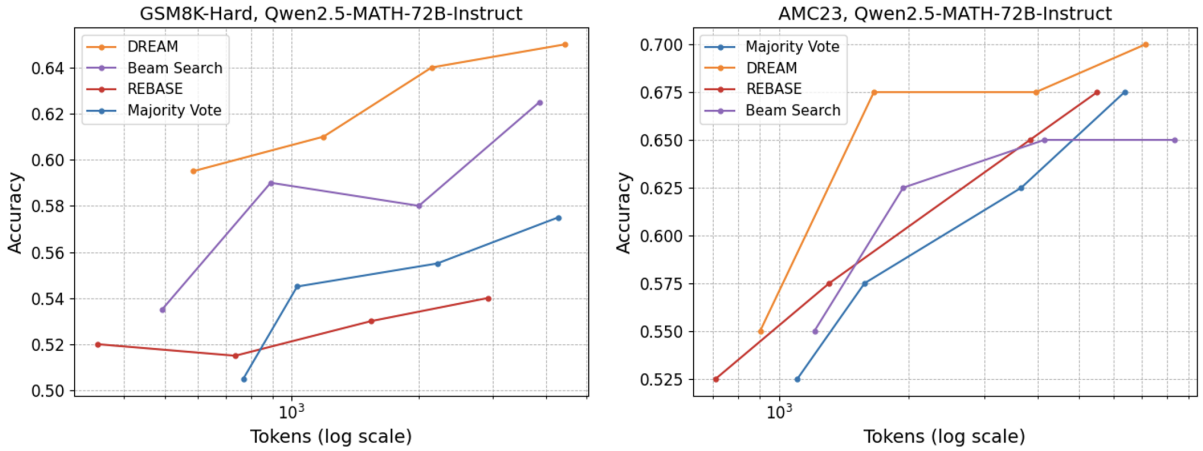

Figure 6: Accuracy-Efficiency Trade-off on Hard Datasets with Qwen2.5-MATH-72B-Instruct

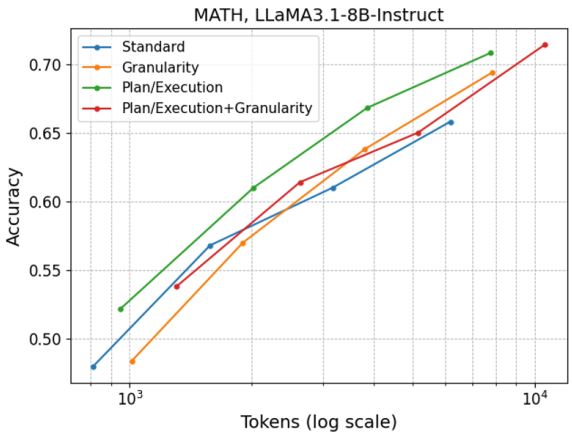

Figure 7: Comparison of accuracy-efficiency trade-off of the four CoT formats

composition rather than simply splitting steps into smaller units, we compare all the four CoT variants shown in the above example using DREAM under different beam widths. The numerical results are provided in Table 12 and visualized in Figure 7.

Based on the results, we have the following observations. First, when comparing entries within each row of the table i.e., under the same searching width, we observe both increasing granularity and applying plan–execution decomposition can improve accuracy, but both approaches also require more tokens and therefore higher computation. However, the accuracy–efficiency trade-off shown in Figure 1 reveals a clear difference: there is no obvious gain from finer-grained steps in terms of accuracy-efficiency trade-off, whereas the gain from plan–execution decomposition is obvious.

Second, when the beam width is fixed, combining increased granularity with plan–execution decomposition leads to higher accuracy than using plan–execution alone. However, this combined strategy substantially increases the number of generated tokens and makes the reasoning trace significantly more verbose. As a consequence, the computational cost grows faster than the improvement in accuracy. When examining the accuracy–efficiency trade-off, we do not observe advantages beyond using plan–execution decomposition alone.

In summary, the results confirm that the benefit of DREAM does not come from simply making the reasoning steps more fine-grained, but from the structural decomposition that separates high-level subgoal planning from low-level execution, enabling more effective exploration and pruning and ultimately yielding a better accuracy–efficiency trade-off.

Table 12: Comparison of four CoT formats on the MATH dataset with LLaMA3.1-8B-Instruct

Standard		Granularity		Plan / Execution		Plan/Execution + Granularity	
acc	**tokens**	**acc**	**tokens**	**acc**	**tokens**	**acc**	**tokens**
0.480	811.938	0.484	1011.53	0.522	949.49	0.538	1302.44
0.568	1578.09	0.570	1902.53	0.610	2021.16	0.614	2631.79
0.610	3170.16	0.638	3805.05	0.668	3854.31	0.650	5149.69
0.658	6188.42	0.694	7872.97	0.708	7776.57	0.714	10587.93

Table 13: Qwen2.5-MATH-72B-Instruct on GSM8k-Hard

DREAM		Beam		Rebase		Majority Vote	
acc	**tokens**	**acc**	**tokens**	**acc**	**tokens**	**acc**	**tokens**
0.595	584.3	0.535	492.9	0.520	347.8	0.505	768.0
0.610	1190.0	0.590	891.9	0.515	735.2	0.545	1030.3
0.640	2146.1	0.580	2001.4	0.530	1541.0	0.555	2210.8
0.650	4436.8	0.625	3863.7	0.540	2911.2	0.575	4281.1

Table 14: Qwen2.5-MATH-72B-Instruct on AMC23

DREAM		Beam Search		REBASE		Majority Vote	
acc	**tokens**	**acc**	**tokens**	**acc**	**tokens**	**acc**	**tokens**
0.55	901.7	0.550	1206.6	0.525	710.4	0.525	1101.4
0.675	1660.2	0.625	1943.3	0.575	1306.4	0.575	1579.4
0.675	3960.8	0.650	4142.1	0.650	3837.7	0.625	3663.3
0.700	7128.5	0.650	8314.9	0.675	5503.7	0.675	6373.8

K Few-shot prompts for guiding reasoning in the plan–execution format

Table 15: Few-shot Prompt for the GSM8K dataset

Learn from the following examples to answer the given question. For each step, begin by stating the general plan, followed by executing that plan (started with "-"), separated by a colon.
When the answer is finished, end with “The final answer is:".

Q: A father is building a playset for his son and needs to purchase lumber, nails, and fabric. When he started planning the project, the necessary lumber cost $450, the nails cost $30, and the fabric cost $80. However, recent economic inflation has caused the price of lumber to increase by 20%, the price of nails to increase by 10%, and the price of fabric to increase by 5%. In dollars, how much more money will it cost to complete the project now (after inflation) than it would have when the father began planning?
Step 1: Find the new cost of lumber:
- The lumber originally cost $450, and with a 20% increase it becomes $\$450 \times 1.20 = 540\$$.
Step 2: Find the new cost of nails:
- The nails originally cost $30, and with a 10% increase they become $30 \times 1.10 = 33$.
Step 3: Find the new cost of fabric:
- The fabric originally cost $80, and with a 5% increase it becomes $80 \times 1.05 = 84$.
Step 4: Find the total original cost:
- Before inflation, the total was $450 + 30 + 80 = 560$.
Step 5: Find the total new cost:
- After inflation, the total is $540 + 33 + 84 = 657$.
Step 6: Compare the costs:
- The difference between the new and old total is $657 - 560 = 97$. The final answer is: 97.

Q: There are five times as many swordfish as pufferfish in an aquarium exhibit. If there are 90 fish total, how many pufferfish are there?
Step 1: Express swordfish in terms of pufferfish:
- If the number of pufferfish is x, then the number of swordfish is $5x$.
Step 2: Write the total number of fish:
- The total is $x + 5x = 6x$.
Step 3: Set up the equation:
- Since the total is 90, we have $6x = 90$.
Step 4: Solve for x:
- Dividing both sides by 6 gives $x = 15$. The final answer is: 15.

Q: Mrs. Johnson’s class raised $2300, which is twice what Mrs. Sutton’s class raised. If Mrs. Sutton’s class raised 8 times less than Miss Rollin’s class, and Miss Rollin’s class raised one-third of the total, how much money did the school raise after deducting 2% fees?
Step 1: Find Mrs. Sutton’s amount:
- Since Mrs. Johnson’s $2300 is twice Sutton’s, Sutton’s amount is $2300 / 2 = 1150$.
Step 2: Find Miss Rollin’s amount:
- Rollin’s class raised 8 times more than Sutton’s, so $1150 \times 8 = 9200$.
Step 3: Find the school’s total before fees:
- Since Rollin’s amount is one-third of the total, the total is $9200 \times 3 = 27600$.
Step 4: Calculate the deduction:
- 2% of 27600 is $27600 \times 0.02 = 552$.
Step 5: Find the net amount:
- Subtracting the deduction gives $27600 - 552 = 27048$. The final answer is: 27048.

Q: John buys 500 newspapers. Each newspaper sells for $2. He sells 80% of them. He buys them all for 75% less than the price at which he sells them. How much profit does he make?
Step 1: Determine how many he sells:
- He sells 80% of 500, which is $0.8 \times 500 = 400$.
Step 2: Determine the selling price:
- Each newspaper sells for $2.
Step 3: Compute revenue:
- Selling 400 at $2 each gives $400 \times 2 = 800$.
Step 4: Determine purchase price per paper:
- He bought them 75% cheaper, meaning at 25% of 2, which is 0.50.
Step 5: Compute total cost:
- Buying 500 at $0.50 each costs $500 \times 0.50 = 250$.
Step 6: Calculate profit:
- Revenue minus cost is $800 - 250 = 550$. The final answer is: 550.

Continue:

Q: A house's living room, dining room and kitchen take up 1,000 sq ft. The house also has a guest bedroom and a master bedroom suite. If the guest bedroom is a quarter of the size of the master bedroom suite, and the house is 2,300 sq ft total, how large is the master bedroom suite?
Step 1: Find how much space is left for the bedrooms:
- The house is 2,300 sq ft total, and 1,000 sq ft is already used by the living room, dining room, and kitchen. So the bedrooms must occupy $2300 - 1000 = 1300$ sq ft.
Step 2: Express the guest bedroom in terms of the master bedroom:
- If the master bedroom is x sq ft, the guest bedroom is one quarter of that, or $x/4$.
Step 3: Write the total space of both bedrooms:
- Together the master and guest bedrooms occupy $x + x/4 = (5/4)x$.
Step 4: Solve for the master bedroom size:
- Since the two bedrooms together are 1,300 sq ft, we have $(5/4)x = 1300$.
Solving gives $x = (1300 \times 4)/5 = 1040$. The final answer is: 1040.

Q: Carter usually bakes 6 cheesecakes, 5 muffins, and 8 red velvet cakes regularly for a week. For this week he was able to bake triple the number of cheesecakes,muffins, chocolate moist cakes, and red velvet cakes. How much more cakes was Carter able to bake for this week?
Step 1: Calculate Carter's usual total:
- Normally he bakes $6 + 5 + 8 = 19$ cakes per week.
Step 2: Find his cheesecakes this week:
- Tripling 6 cheesecakes gives $6 \times 3 = 18$.
Step 3: Find his muffins this week:
- Tripling 5 muffins gives $5 \times 3 = 15$.
Step 4: Find his red velvet cakes this week:
- Tripling 8 red velvet cakes gives $8 \times 3 = 24$.
Step 5: Add up the new totals:
- Altogether he baked $18 + 15 + 24 = 57$ cakes this week.
Step 6: Compare to his usual:
- Compared to 19 cakes normally, the increase is $57 - 19 = 38$. The final answer is: 38.

Q: Four adults with 32 teeth went to the dentist. The first had 1/4 removed, the second 3/8 removed, the third half removed, and the fourth 4 removed. What's the total number of teeth removed?
Step 1: Find the first person's removal:
- One-quarter of 32 teeth is $32 \times 1/4 = 8$.
Step 2: Find the second person's removal:
- Three-eighths of 32 teeth is $32 \times 3/8 = 12$.
Step 3: Find the third person's removal:
- Half of 32 teeth is $32 \times 1/2 = 16$.
Step 4: Note the fourth person's removal:
- This was directly stated as 4 teeth.
Step 5: Add all removals together:
- $8 + 12 + 16 + 4 = 40$ teeth removed in total. The final answer is: 40.

Q: Joy fosters dogs. The mom dog eats 1.5 cups of food three times a day. The puppies (5 total) each eat 0.5 cups twice a day. How much food will Joy need for the next 6 days?
Step 1: Calculate the mom's daily food:
- She eats 1.5 cups at each of 3 meals, so $1.5 \times 3 = 4.5$ cups per day.
Step 2: Calculate one puppy's daily food:
- Each puppy eats 0.5 cups at 2 meals, so $0.5 \times 2 = 1$ cup per day.
Step 3: Calculate all puppies' daily food:
- With 5 puppies, their total is $1 \times 5 = 5$ cups per day.
Step 4: Find the combined daily food:
- Adding the mom and puppies gives $4.5 + 5 = 9.5$ cups per day.
Step 5: Find the total for 6 days: Over 6 days,
- Joy needs $9.5 \times 6 = 57$ cups. The final answer is: 57.

Q: [New Question to be Solved]

Table 16: Few-show prompt for the MATH dataset

Learn from the following examples to answer the given question. For each step, begin by stating the general plan, followed by executing that plan (started with "-"), separated by a colon.
When the answer is finished, end with “The final answer is:".

Q: Karl was attempting to calculate economic figures. He found the following equation to be true: $fp - w = 10000$. If $f = 5$and $w = 5 + 125i$, what is p?
A: Step 1: Write down the expression for fp in terms of w:
- $fp = 10000 + w$.
Step 2: Write down the expression for fp using the value of w:
- Substitute $w = 5 + 125i$into the equation, we get $fp = 10000 + 5 + 125i = 10005 + 125i$.
Step 3: Calculate the value of p:
- Since $f = 5$, we compute $p = \frac{fp}{f} = \frac{10005+125i}{5} = 2001 + 25i$. The final answer is: $2001 + 25i$.

Q: The roots of the equation $2x^2 - mx + n = 0$sum to 6and multiply to 10.
What is the value of $m + n$?
A: Step 1: Identify the relationships between the roots and coefficients of a quadratic:
- For a quadratic $ax^2 + bx + c = 0$, the sum of the roots is $-\frac{b}{a}$and the product is $\frac{c}{a}$.
Step 2: Use the sum of roots to find m:
- Here, $a = 2$and $b = -m$, so the sum is $-\frac{-m}{2} = \frac{m}{2}$.
Setting this equal to 6gives $\frac{m}{2} = 6 \Rightarrow m = 12$. Step 3: Use the product of roots to find n:
- The product is $\frac{n}{2}$, and it's given as 10. So $\frac{n}{2} = 10 \Rightarrow n = 20$.
Step 4: Add $m$and n:
- $m + n = 12 + 20 = 32$. The final answer is: 32.

Question: If the two roots of the quadratic $7x^2 + 3x + k$are $\frac{-3\pm i\sqrt{299}}{14}$, what is k?
A: Step 1: Recall the quadratic formula:
- The roots of $ax^2 + bx + c = 0$are given by $\frac{-b\pm\sqrt{b^2-4ac}}{2a}$.
Step 2: Identify coefficients:
- From the equation $7x^2 + 3x + k$, we have $a = 7$, $b = 3$, $c = k$.
Step 3: Write the discriminant in terms of k:
- $b^2 - 4ac = 3^2 - 4 \cdot 7 \cdot k = 9 - 28k$.
Step 4: Use the root form to determine the discriminant:
- Since the roots are $\frac{-3\pm i\sqrt{299}}{14}$, the discriminant is -299. So $9 - 28k = -299 \Rightarrow 28k = 308 \Rightarrow k = 11$.
The final answer is: 11.

Question: Rationalize the denominator of $\frac{1}{\sqrt{2}+\sqrt{3}+\sqrt{7}}$, and write your answer in the form $\frac{A\sqrt{2}+B\sqrt{3}+C\sqrt{7}+D\sqrt{E}}{F}$. What is $A + B + C + D + E + F$?
A: Step 1: Multiply by a conjugate to simplify:
- Multiply numerator and denominator by $(\sqrt{2} + \sqrt{3}) - \sqrt{7}$.
Step 2: Simplify the denominator:
- $(\sqrt{2} + \sqrt{3})^2 = 2 + 3 + 2\sqrt{6} = 5 + 2\sqrt{6}$, and the full denominator becomes $(5 + 2\sqrt{6}) - 7 = -2 + 2\sqrt{6}$.
Step 3: Multiply by the conjugate of the new denominator:
- Multiply numerator and denominator by $-2 - 2\sqrt{6}$: $\frac{\sqrt{2}+\sqrt{3}-\sqrt{7}}{-2+2\sqrt{6}} \cdot \frac{-2-2\sqrt{6}}{-2-2\sqrt{6}} = \frac{4\sqrt{2}+3\sqrt{3}-\sqrt{7}-\sqrt{42}}{10}$.
Step 4: Identify constants and sum:
- $A = 4, B = 3, C = -1, D = -1, E = 42, F = 10$,
so the total is: $A + B + C + D + E + F = 4 + 3 - 1 - 1 + 42 + 10 = 57$. The final answer is: 57.

Question: For what values of $b$is -2 not in the range of the function $f(x) = x^2 + bx + 2$?
Express your answer in interval notation.
A: Step 1: Understand when $-2$is in the range:
- $-2$is in the range if there exists some $x$such that $f(x) = -2$.

Continue:

Step 2: Set up the equation:
- $f(x) = -2 \Rightarrow x^2 + bx + 2 = -2 \Rightarrow x^2 + bx + 4 = 0$.
Step 3: Determine when the equation has no real solutions:
- A quadratic has no real solutions when the discriminant is negative.
Step 4: Solve for values of b:
- $b^2 - 16 < 0 \Rightarrow b^2 < 16 \Rightarrow -4 < b < 4$. The final answer is: $(-4, 4)$.

Question: What is the value of $\sqrt[3]{3^5 + 3^5 + 3^5}$?
A: Step 1: Calculate 3^5:
- $3^5 = 243$.
Step 2: Add three copies of 3^5:
- $3^5 + 3^5 + 3^5 = 243 + 243 + 243 = 729$.
Step 3: Take the cube root:
- $\sqrt[3]{729} = 9$since $9^3 = 729$.
The final answer is: 9.

Q: [New Question to be Solved]