DISC: DYNAMIC DECOMPOSITION IMPROVES LLM INFERENCE SCALING

Anonymous authors
Paper under double-blind review

ABSTRACT

Inference scaling methods often rely on decomposing problems into steps (or groups of tokens), followed by sampling and selecting the best next steps. However, these steps and their sizes are often predetermined or manually designed based on domain knowledge. We propose dynamic decomposition, a method that adaptively and automatically fractions solution and reasoning traces into manageable steps during inference. By more effectively allocating compute – particularly through subdividing challenging steps and prioritizing their sampling – dynamic decomposition significantly improves inference efficiency. Experiments on benchmarks such as APPS, MATH, and LiveCodeBench demonstrate that dynamic decomposition outperforms static approaches, including token-level, sentence-level, and single-step decompositions. These findings highlight the potential of dynamic decomposition to improve a wide range of inference scaling techniques.

1 INTRODUCTION

Scaling inference efficiency remains a fundamental challenge for large language models (LLMs). Many existing approaches improve inference by decomposing problems into smaller steps and systematically exploring different solutions (Feng et al., 2023; Zeng et al., 2024; Wu et al., 2024; Nori et al., 2024; Snell et al., 2024; Brown et al., 2024; Gandhi et al., 2024; Lee et al., 2025; Light et al., 2024a; Wang et al., 2025).

Some decomposition methods rely on domain-specific heuristics and hand-crafted rules (Yao et al., 2024; Zelikman et al., 2023; Zhou et al., 2022). However, manually partitioning problems or designing task-specific heuristics is costly and lacks generalization. Moreover, identifying critical steps for an LLM can be non-trivial for humans. As shown in Sec. 3.5, LLMs may assign importance to seemingly trivial words (e.g., therefore or which), which, while counterintuitive to humans, play a crucial role in autoregressive generation (Lin et al., 2025). Other approaches employ fixed, uniform step sizes, such as token- or sentence-level decomposition (Feng et al., 2023; Guo et al., 2025). All these methods rely on **static decomposition strategies**, where step sizes are predefined or determined via heuristics. Such rigidity wastes compute on steps that are easy for the LLM (but potentially difficult for humans) while undersampling more challenging steps.

To overcome these limitations, we propose DISC (Dynamic decomposition Improves Scaling Compute), a recursive inference algorithm that dynamically partitions solution steps based on difficulty. Unlike prior methods, DISC **adapts decomposition granularity** during inference based on both the available budget and problem complexity, ensuring finer granularity for more difficult steps. By leveraging the autoregressive nature of LLMs, DISC efficiently **locates difficult steps** through binary partitioning, focusing compute on challenging regions rather than wasting resources on trivial steps. DISC is *generalizable* and requires *no human supervision, domain-specific heuristics, prompt engineering, or process annotations*, making it widely applicable across tasks.

Our main contributions are:
- We introduce DISC, a method for recursive partitioning and decomposing solutions during inference *without human supervision, domain-specific heuristics, or process reward models*.
- We demonstrate how DISC integrates decomposition with inference-time search, **allocating compute to high-impact, difficult steps**.
- We show that DISC improves inference scaling in terms of both **sample efficiency** and **token efficiency**.

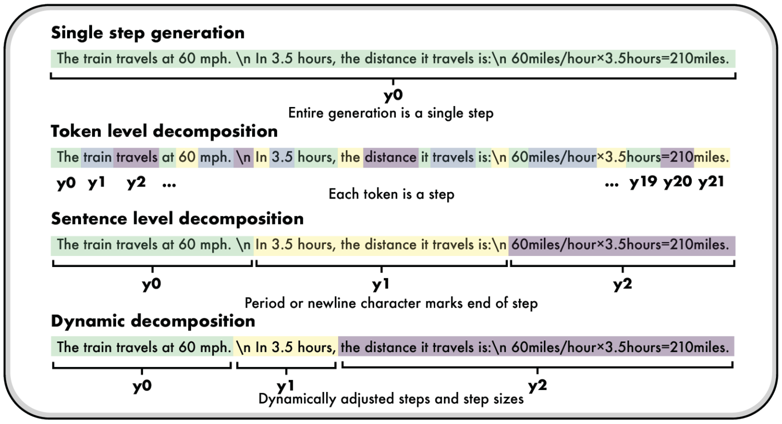

Figure 1: **Comparison of different automatic decomposition methods** based on step size determination.

- We provide insights into how LLMs reason and plan by identifying **critical steps** in their generation process.

2 Preliminaries

2.1 Problem Setting

We consider a reasoning and code generation setting where a dataset $\mathcal{X} = \{\boldsymbol{x}^{(i)}\}_{i=1}^{N}$ consists of problem prompts $\boldsymbol{x}$, and a reward model $R : \mathcal{X} \cdot \mathcal{Y} \to [0, 1]$ evaluates generated solutions $\boldsymbol{y} \in \mathcal{Y}$. This includes program synthesis, where correctness is verified using ground-truth tests (Chen et al., 2021; Austin et al., 2021), and mathematical reasoning, where solutions are validated numerically (Hendrycks et al., 2021a; Cobbe et al., 2021). The reward model can be a ground-truth verifier, a trained heuristic (Zhang et al., 2024), self-consistency (Wang et al., 2023a), or an LLM-as-a-judge (Zheng et al., 2023). Since our focus is on step decomposition rather than verification, we use the ground-truth reward model where available. We assume access to a pretrained language model π that generates text autoregressively. A generated response $\boldsymbol{y}$ consists of both the final solution and the reasoning chain leading to it, and can be represented as a sequence of tokens $\boldsymbol{y} = (y_0, ..., y_{L_{\boldsymbol{y}}})$. Additionally, solutions can be partitioned into **solution steps** $\boldsymbol{y} = (\boldsymbol{y}_0, ..., \boldsymbol{y}_K)$, where each step $\boldsymbol{y}_i$ is a contiguous string of tokens. A **partial solution** up to step k is defined as $\boldsymbol{y}_{1...k} := \boldsymbol{y}_1 \cdot \boldsymbol{y}_2 \cdot ... \cdot \boldsymbol{y}_k$, and its **rollout** or **completion**, denoted $\boldsymbol{y}_{1...k+}$, is the continuation generated by π until an end-of-sequence token (EOS). The **size** of a solution step, $|\boldsymbol{y}_i|$, refers to its length in tokens or characters.

2.2 Prior Automatic Decomposition Methods

Single-step generation. In a single-step generation, the entire solution is generated in one pass from the prompt to the EOS token, treating it as a single action. This approach underlies the widely used inference scaling method **best of n** (BoN) (Cobbe et al., 2021; Lightman et al., 2023; Snell et al., 2024; Liang et al., 2024), where n complete solutions are sampled, and the highest-scoring one is selected. Single-step generation also plays a role in alignment and fine-tuning methods such as **DPO** (Rafailov et al., 2024) and **RLOO** (Ahmadian et al., 2024).

Token-level decomposition. At the opposite end of the spectrum, token-level decomposition treats each atomic token as an individual step. While this approach dramatically increases search complexity, it enables fine-grained search that can yield higher performance gains given sufficient compute (Feng et al., 2023).

Newline and sentence-level decomposition. A commonly used decomposition method segments LLM generations into sentences or lines based on delimiters such as periods or newlines (Hao et al., 2023; Feng et al., 2023; Yao et al., 2024). Typically, each newline corresponds to a new paragraph, equation, or line of code, which often encapsulates a distinct reasoning step.

See App. C for more discussion of prior methods and related works.

Problem: Automatic and scalable decomposition

Existing decomposition methods are task-specific, manual, and static, limiting their adaptability and scalability.

2.3 Step Sampling

Inference-time scaling methods must balance exploration at the current step with exploration of future steps. We implement a simple dynamic sampling process, referred to as **negative binomial sampling**, where we continue sampling completions until the *sum of their rewards exceeds a predefined threshold* σ. More formally, the number of samples M drawn from a partial solution $\boldsymbol{y}_{1\ldots k}$ is the smallest integer satisfying $\sum_{i=1}^{M} R(\boldsymbol{x} \cdot \boldsymbol{y}^{(i)}_{1\ldots k+}) \geq \sigma$. Here, $\boldsymbol{y}^{(i)}_{1\ldots k+}$ represents the i-th sampled completion from the partial solution. This process ensures efficient allocation of compute by dynamically adjusting the number of samples per step. It achieves this by either: (a) *continuing to sample until a sufficiently high-reward completion is found*, or (b) *stopping early when additional sampling is unlikely to yield significant improvements*, thus redirecting the compute to future steps. The stopping criterion is governed by σ: when accumulated reward from completions surpasses σ, the method assumes further sampling is unnecessary. For a fair comparison, we apply this sampling method *uniformly across all decomposition methods* in our experiments.

2.4 Inference Scaling Methods and Decomposition

Since our study focuses on decomposition rather than search, we primarily use **greedy step search** as the search method. In greedy step search, multiple candidate steps are sampled at each iteration, but only the highest-scoring step is retained, while the rest are discarded. The process then repeats, conditioning future steps on the best step found so far. We also perform ablation studies comparing **Monte Carlo Tree Search** (MCTS) (Feng et al., 2023; Light et al., 2024b) and **beam search** (Xie et al., 2024), two commonly used inference scaling methods. These comparisons, presented in Sec. 4.2, highlight how different search strategies interact with decomposition. Additional details on MCTS and beam search are provided in App. F.

3 Methodology

3.1 DISC Algorithm

The DISC algorithm employs recursive binary decomposition to iteratively break down complex solutions into smaller, more manageable steps. Given a problem prompt $\boldsymbol{x}$, the algorithm outputs a decomposition of a solution, $\boldsymbol{y} = (\boldsymbol{y}_1, \boldsymbol{y}_2, \ldots, \boldsymbol{y}_K)$, such that the concatenation $\boldsymbol{y}_{1\ldots K}$ forms a complete solution to $\boldsymbol{x}$.

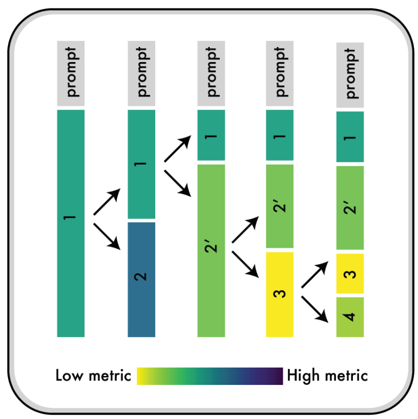

Figure 2: DISC recursively partitions the most challenging step – i.e., the one with the lowest priority metric h – to progressively refine and find the optimal solution.

The algorithm operates in two key stages:

1. Solution sampling. Starting from $\boldsymbol{x}$ and $\boldsymbol{y}_0 = \emptyset$, the algorithm generates complete solutions $\boldsymbol{y}_k \sim \pi(\cdot|\boldsymbol{x} \cdot \boldsymbol{y}_{1\ldots(1-k)})$ using a policy π like in single step generation. The best solution, $\boldsymbol{y}^*_k$, is selected based on the reward model R.

2. Recursive partitioning. The selected solution $\boldsymbol{y}^*_k$ is partitioned into two segments, $\boldsymbol{y}^*_k = \boldsymbol{y}^*_a \cdot \boldsymbol{y}^*_b$, based on a predefined partition fraction α, where $|\boldsymbol{y}^*_a| \approx \alpha|\boldsymbol{y}^*_k|$. For each part, a priority metric h is estimated: $\widehat{h}(\boldsymbol{y}^*_a|\boldsymbol{x} \cdot \boldsymbol{y}_{1\ldots(k-1)})$ and $\widehat{h}(\boldsymbol{y}^*_b|\boldsymbol{x} \cdot \boldsymbol{y}^*_{1\ldots a})$, usually through rollouts of the step using π. The part with the lower priority is further partitioned.

- If $\widehat{h}(\boldsymbol{y}^*_a|\boldsymbol{x} \cdot \boldsymbol{y}_{1\ldots(k-1)}) \geq \widehat{h}(\boldsymbol{y}^*_b|\boldsymbol{x} \cdot \boldsymbol{y}^*_{1\ldots a})$, additional samples are sampled for $\boldsymbol{y}^*_b$, with the process repeating on the new best solution, $\boldsymbol{y}'^*_b$.
- Conversely, if $\widehat{h}(\boldsymbol{y}^*_a|\boldsymbol{x} \cdot \boldsymbol{y}_{1\ldots(k-1)}) < \widehat{h}(\boldsymbol{y}^*_b|\boldsymbol{x} \cdot \boldsymbol{y}^*_{1\ldots a})$, the first segment $\boldsymbol{y}^*_a$ is further partitioned. The first step corresponds to the α fraction of $\boldsymbol{y}'^*_a$, with the remaining part of the full solution forming the second step.

This recursive process is illustrated in Fig. 2. The pseudocode for DISC is provided in Alg. 1, with an annotated Python implementation in App. A.

The **priority metric** h serves as the central heuristic for determining which solution steps to prioritize. It estimates the "difficulty" or "potential for improvement" of a step $\boldsymbol{y}_k$ given the context $\boldsymbol{x} \cdot \boldsymbol{y}_{1\ldots(k-1)}$, computed via rollouts of the policy π. Specifically, $h(\boldsymbol{y}_k | \boldsymbol{x} \cdot \boldsymbol{y}_{1\ldots(k-1)})$ is estimated by sampling continuations and evaluating their outcomes.

In practice, estimating h and generating new samples occur simultaneously, as both rely on rollout-based computations (Sec. 3.2). Unlike standard decomposition methods, DISC does not process steps in strict temporal order, resembling goal-directed planning (Parascandolo et al., 2020) and *backtracking*.

Intuition and Benefits. DISC partitions difficult steps into smaller, simpler sub-steps, allocating additional resources to refine them. This approach is particularly effective for inference scaling, where computing must be used judiciously. A binary decomposition strategy enables fast identification of difficult or high-potential steps.

Key Insight: Recursive partitioning

Top-down, recursive partitioning means that we can both efficiently locate critical steps and also dynamically determine step sizes based on our budget.

Dynamic Compute Allocation. A key advantage of DISC is its ability to prioritize challenging or high-potential steps, improving solution quality while minimizing compute waste. By iteratively refining steps with low priority scores, DISC adaptively allocates more resources to difficult steps and less to simpler ones, optimizing inference efficiency.

Key Insight: Adaptive compute allocation

DISC dynamically allocates inference compute to harder steps, optimizing solution quality and resource efficiency.

3.2 Priority Metric

We consider two intuitive priority metrics for step selection: **Q-value priority** and **Z-score priority**, which are visualized in Fig. 4.

Q-value based priority (DISC-Q). Given a partial solution $\boldsymbol{y}^*_{1\ldots k} = \boldsymbol{y}^*_{1\ldots(k-1)} \cdot \boldsymbol{y}^*_a \cdot \boldsymbol{y}^*_b$, we aim to prioritize either $\boldsymbol{y}^*_a$ or $\boldsymbol{y}^*_b$ for refinement. The core intuition behind Q-value prioritization is that *steps with lower Q-values indicate areas needing refinement*, directing compute toward the most challenging parts of the solution. More formally, we define the Q-priority metric under policy π as:

$$h_Q(\boldsymbol{y}^*_k \mid \boldsymbol{x} \cdot \boldsymbol{y}_{1\ldots(k-1)}) = \mathbb{E}_{\boldsymbol{y}_k}\left[Q^\pi(\boldsymbol{y}_k \mid \boldsymbol{x} \cdot \boldsymbol{y}_{1\ldots(k-1)})\right].$$

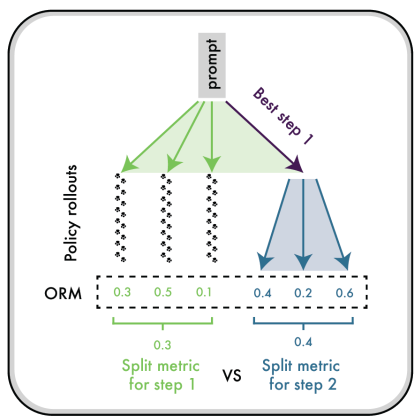

Figure 3: **Priority metric estimation.** We can estimate the priority metric $\widehat{h}$ using Monte Carlo rollouts of the LLM policy π for each step.

Here, $\boldsymbol{y}_k$ represents alternative steps sampled from π, and $Q^\pi(\boldsymbol{y}_k \mid \boldsymbol{x} \cdot \boldsymbol{y}_{1\ldots(k-1)})$ denotes the Q-value of $\boldsymbol{y}_k$, conditioned on the partial solution $\boldsymbol{x} \cdot \boldsymbol{y}_{1\ldots(k-1)}$. Equivalently, $h_Q(\boldsymbol{y}^*_k)$ can be interpreted as the value function $V^\pi(\boldsymbol{x} \cdot \boldsymbol{y}_{1\ldots(k-1)})$, where V^π represents the expected reward achievable from the given partial solution. The expectation is taken over sampled candidates $\boldsymbol{y}_k \sim \pi(\cdot \mid \boldsymbol{x} \cdot \boldsymbol{y}_{1\ldots(k-1)})$, constrained by $|\boldsymbol{y}_k| = |\boldsymbol{y}^*_k|$. This metric helps identify *difficult steps* that the LLM is likely to get wrong, guiding partitioning toward the most critical refinements.

To estimate h_Q for the second step $\boldsymbol{y}^*_b$, we sample $\boldsymbol{y}_b$ from π, generate rollouts $\boldsymbol{y}_{(k+1)+}$, compute rewards, and average the outcomes. For the previous step $\boldsymbol{y}^*_a$, we reuse rollouts from earlier partitioning, as the mean of these previously generated samples provides an unbiased estimate of the Q-priority metric:

$$h_Q(\boldsymbol{y}^*_a \mid \boldsymbol{x} \cdot \boldsymbol{y}^*_{1\ldots(k-1)}) = \mathbb{E}\left[R(\boldsymbol{y}^*_{1\ldots(k-1)+})\right].$$

By leveraging existing rollouts, DISC avoids redundant sampling, improving computational efficiency. Once the rollouts for $\boldsymbol{y}^*_{(k+1)}$ are available, the best completion $\boldsymbol{y}^*_{(k+1)+}$ is selected as the next step to partition. This dual use of rollouts optimizes both metric estimation and inference sampling.

Takeaway: Combining step priority estimation and inference-time search

By integrating LLM policy sampling for both metric estimation and search, we can significantly enhance computational efficiency.

Z-score based priority (DISC-Z). To allocate more compute to steps with *higher potential for improvement*, we estimate the probability of sampling a better step given existing samples. Assuming the Q-values of sampled steps follow a normal distribution, we model this probability using the cumulative distribution function (CDF). Given a mean μ_r and standard deviation σ_r of sampled Q-values, the probability of sampling better than the best-observed step $\boldsymbol{y}^*_k$ with Q-value q^* is:

$$1 - \mathrm{CDF}\left(\frac{q^* - \mu_r}{\sigma_r}\right) = 1 - \mathrm{CDF}(z^*),$$

where z^* is the Z-score. Since the CDF is monotonic, we compare steps based on their Z-scores.

We formally define the Z-score priority metric as:

$$h_Z(\boldsymbol{y}^*_k \mid \boldsymbol{x} \cdot \boldsymbol{y}_{1\ldots(k-1)}) = \frac{q^* - \mathbb{E}[Q^\pi(\boldsymbol{y}_k \mid \boldsymbol{x} \cdot \boldsymbol{y}_{1\ldots(k-1)})]}{\mathrm{Std}[Q^\pi(\boldsymbol{y}_k \mid \boldsymbol{x} \cdot \boldsymbol{y}_{1\ldots(k-1)})]}.$$

Since $q^* - \mu_r$ represents the advantage of step $\boldsymbol{y}^*_k$, the Z-score metric can be interpreted as a *standard deviation-scaled advantage*. Lower h_Z values indicate steps with greater room for improvement, guiding decomposition toward those with higher variance in performance.

3.3 DISC and Search Methods

DISC can also be used to enhance **Monte Carlo Tree Search (MCTS)** and other inference scaling and search methods. Recall that in our partition step, we greedily partition the best solution step $\boldsymbol{y}^*$. Instead of greedily partitioning the best step, we can partition the top k best steps instead, and select which step to partition using the upper confidence tree (UCT) formula. We can also use **beam search** to prune out steps we do not want to partition further. We explain MCTS and beam search in detail in App. F and present results of combining DISC with search in Sec. 4.9.

Takeaway: Dynamic step sizes can improve search

DISC can enhance search based inference scaling methods by determining what step size to search across.

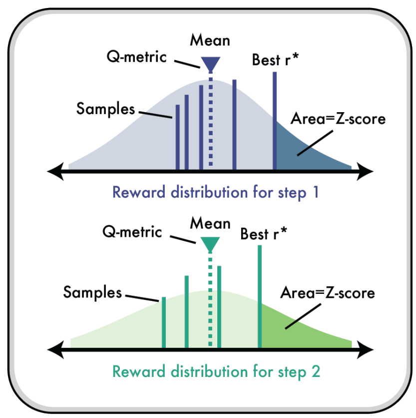

Figure 4: **Different priority metrics.** Sampled rewards of different steps can be visualized as a distribution.

3.4 A Motivating Example on DISC-Z

We use the Wiener process $W(t)$ as an example where there are intractably many actions and steps. Suppose we start at $t = 0$ with $W(0) = 0$. At each round k, the algorithm can choose one of the two options:

1. samples a trajectory and observe the final value $W(T)$ at time $t = T$, as the reward signal. Denote the whole trajectory as $w_k(\cdot)$.
2. chooses one trajectory from the previous rounds (denoted as $w_s(t)$ for round s), and time t_0; then sample a trajectory at $t = t_0$ with $W(t_0) = w_s(t_0)$. Denote the concatenated trajectory as $w_k(\cdot)$ with $w_k(t) = w_s(t)$ when $t \leq t_0$.

Note that we are only able to observe the final reward $W(t)$. At any intermediate time $t \in (0, T)$, the current value $W(t)$ is not observable. The goal is to design an algorithm that can reach the highest reward among the K trajectories. Formally speaking, we aim to maximize the maximum:

$$\max_{k \in K} w_k(T).$$

One naive solution is to call option 1 for K times and return the best-of-K reward, each following:

$$W(T) \sim \mathcal{N}(0, T).$$

Alternatively, suppose there is a promising path $w(\cdot)$ with a high final reward $w(T) = R$. It is natural to consider starting at some midpoint αT $(0 < \alpha < 1)$ and perform more completions to obtain an even higher reward than R. The reward distribution sampled this way is

$$W'(T) \sim \mathcal{N}(w(\alpha T), (1 - \alpha)T).$$

The remaining question is which α we should choose. One option is to maximize the probability that the newly sampled reward is higher than R:

$$\mathbb{P}(W'(T) > R) = 1 - \Phi\left(\frac{R - w(\alpha T)}{\sqrt{(1-\alpha)T}}\right).$$

3.5 Example Decomposition

With sample budget 100, the decomposition of a MATH problem is as follows, where color indicates the value of the priority metric h of each step (yellow low, purple high).

DISC example decomposition

Let the length of the rectangle be l and the width of the rectangle be w. Since the perimeter of the rectangle is 24 inches, we have that $2l + 2w = 24$, so $l + w = 12$. We wish to maximize the area of the rectangle, which is $A = lw$. Let $l = 12 - w$ and plug into the area:

$$A = (12 - w)w \Rightarrow \qquad A = 12w - w^2$$

Now, we differentiate A with respect to w:

$$A'(w) = 12 - 2w$$

We wish to maximize A, so we set $A'(w) = 0$, and solve for w:

$$12 - 2w = 0 \Rightarrow \qquad w = 6$$

Since $l = 12 - w$, we have that $l = 12 - 6 = 6$. Therefore, the area of the rectangle is $A = lw = 6 \cdot 6 = \boxed{36}$.

Once the LLM generates the first three steps, the rest is easy. Interestingly, 'which' is an important decision point which helps decide how the LLM will complete the solution.

Takeaway: Autoregressive models require autoregressive decomposition

While words such as 'which', 'therefore', etc. may not seem like important steps to humans, they actually represent important steps for autoregressive LLMs which are trained on next token prediction.

4 Experimental Results

4.1 Benchmarks

We evaluate DISC on three benchmarks: **APPS**, **MATH**, and **LiveCodeBench**, to assess its impact on inference scaling for both coding and reasoning. **APPS** (Hendrycks et al., 2021a) consists of 5000 competitive programming problems across three difficulty levels, with the competition-level subset being the hardest. We evaluate on a 200-problem subset due to computational constraints. **MATH** (Hendrycks et al., 2021b) comprises 12,500 math problems. Since the ground-truth verifier provides only binary rewards, we use a pretrained ORM (Xiong et al., 2024), trained via the method in (Wang et al., 2024b), with Llama-3.1-8B-Instruct as the base model. We test on a 500-problem subset (**MATH500**), identical to prior work (Wang et al., 2024b; Lightman et al., 2023). **LiveCodeBench** (Jain et al., 2024) is a continuously updated dataset from Leetcode, AtCoder, and CodeForces, ensuring LLMs have not been exposed to test problems. We evaluate on the 108 problems uploaded between 10/01/2024 and 12/01/2024 to prevent contamination.

4.2 Decomposition Comparison

We compare DISC against three prior decomposition methods from Sec. 2.2: **TokenSplit** (token-level decomposition), **LineSplit** (newline-based decomposition), and **BoN** (treating the entire solution as a single step). Across all benchmarks, DISC achieves superior scaling and performance under both fixed token budgets (Fig. 5) and sample budgets (Fig. 10). We evaluate two key metrics: **Pass@k**,

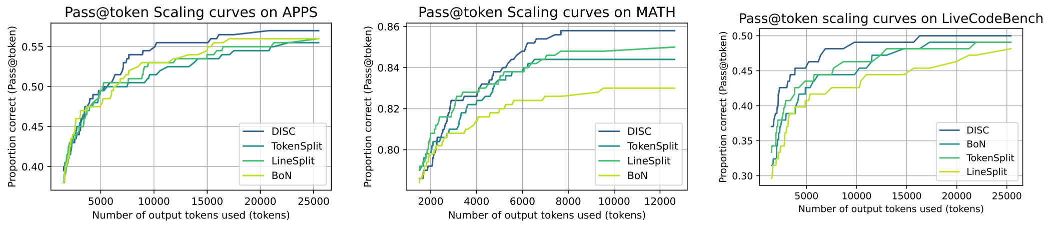

Figure 5: **Token-level comparisons across benchmarks.** (Left) APPS competition level (Middle) MATH500 (Right) LiveCodeBench. DISC achieves superior inference scaling over baselines on all three benchmarks.

the proportion of problems solved within a sample budget k, and **Pass@token**, the proportion solved within a given token budget. Notably, DISC consistently outperforms static decomposition methods on APPS, MATH, and LiveCodeBench (Fig. 5), demonstrating its ability to allocate compute adaptively for improved inference efficiency. Extended results and analyses for each benchmark are provided in App. E.1, E.4, and E.5.

4.3 Decomposition Analysis and Interpretation

Our results strongly indicate that decomposition—whether line-based, token-based, or DISC —improves sample quality. Fig. 6 illustrates how the mean and variance of sampled rewards evolve with the **step number**, which represents the order in which a step is explored. Higher step numbers correspond to deeper search levels, where solutions are partitioned into finer-grained steps. As shown in Fig. 6, increasing step number correlates with higher-quality solutions, demonstrating that finer-grained decomposition improves sample quality. Additionally, Fig. 6 shows that reward variance decreases as step count increases, highlighting how decomposition enhances sampling precision.

Furthermore, DISC achieves better performance with fewer partitions under a fixed sampling budget (Fig. 7). We distinguish between **actual partitions**, the number of steps effectively explored, and **planned partitions**, the number of partitions intended by the method. Token and line split methods generate a large number of planned partitions (Fig. 7) but search over at most 15 steps due to budget constraints. In contrast, DISC dynamically adjusts the number of partitions based on available budget, efficiently identifying and focusing on critical steps.

Takeaway: Decomposition and sample quality

Finer-grained decomposition improves sampled solution quality and reduces reward variance.

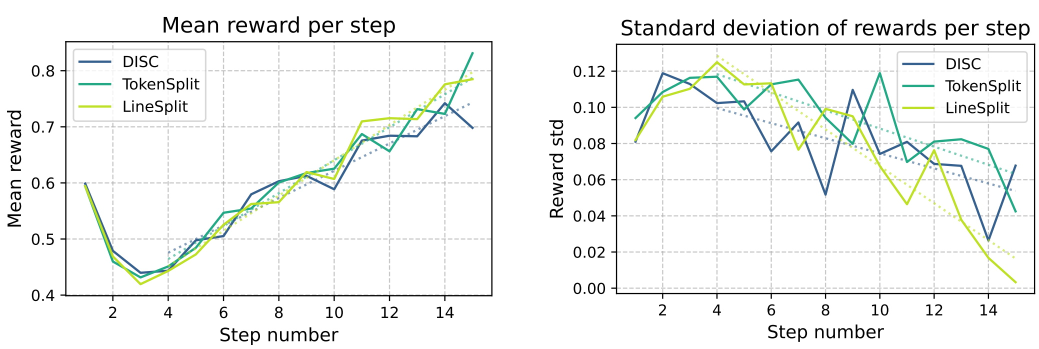

Figure 6: **Analysis of rewards per step on APPS.** (Left) Average reward per step: From step 3 onward, higher step counts strongly correlate with increased average reward, demonstrating the effectiveness of decomposition. The dip between steps 1 and 3 likely occurs because simple problems are solved early, preventing further search. (Right) Standard deviation of rewards per step: Decomposition reduces sampling variance, improving precision at deeper search depths.

4.4 Interaction Between Temperature and DISC

We perform ablation studies to analyze the impact of temperature on DISC. Typically, inference scaling methods achieve optimal performance at temperatures around 0.6–0.8, as increased tempera-

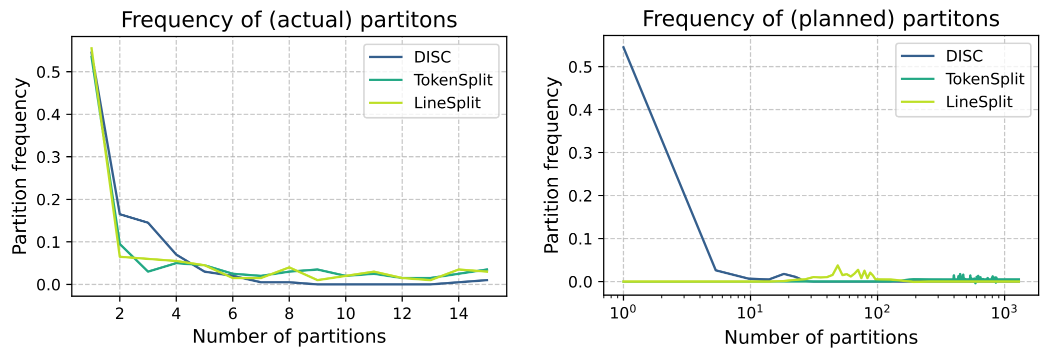

Figure 7: **Comparison of actual and planned partitions on APPS.** DISC outperforms other methods with fewer partitions by efficiently identifying critical steps. Unlike token and line split methods, which plan many partitions but search only a subset, DISC dynamically adjusts partitioning based on budget.

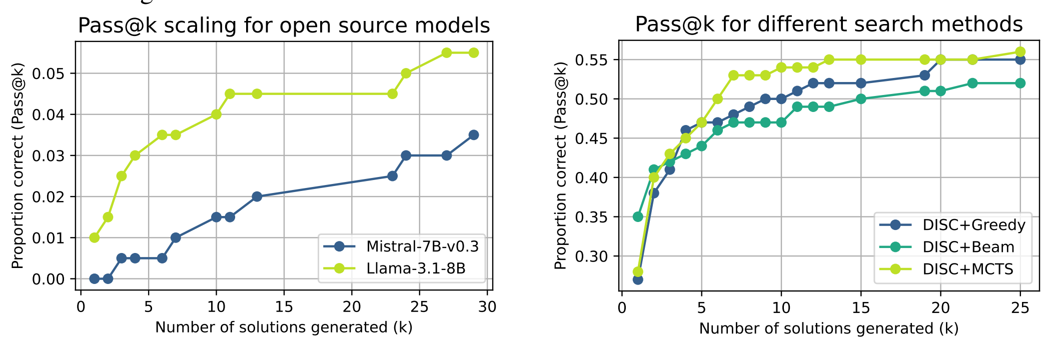

Figure 8: **Pass@k scaling on APPS.** (Left) Open-source models: DISC substantially improves performance across different LLMs, including Llama and Mistral. (Right) Search methods: MCTS scales best, followed by greedy search, then beam search (beam size 2) when combined with DISC on APPS with gpt-4o-mini.

ture promotes sample diversity (Wang et al., 2024a). Surprisingly, however, DISC performs *better at lower temperatures*, as shown in Fig. 9. This trend is in stark contrast to BoN (Fig. 16), where higher temperatures are generally beneficial. We believe this phenomenon arises because DISC depends on accurately estimating the priority metric h at each step. Lower temperatures reduce sample variance, leading to more reliable estimates of h, which in turn improves step selection. This is further supported by Fig. 14, which shows that lower temperatures yield lower standard deviations per step, indicating increased sampling consistency. Additional details and analyses can be found in App. D.1.

4.5 Self-Generated Validation Tests

We also evaluate DISC in a more practical setting where a ground-truth reward model is unavailable for code generation (Chen et al., 2022; 2023b; Zhou et al., 2024). Instead of relying on predefined test cases, we prompt the LLM to generate validation test cases based on the problem prompt. In real-world applications, manually curated ground-truth test cases are often costly to obtain, making self-generated validation a more scalable approach. The results, shown in Fig. 10, indicate that DISC continues to scale better than other methods in this setting. Additional results and details are provided in App. E.3.

4.6 Interaction Between Priority Metric and DISC

We conduct an ablation study to examine how the choice of priority metric affects DISC performance. In addition to the **Q-based** and **Z-based** priority metrics (DISC-Q and DISC-Z) introduced in Sec. 3.2, we evaluate three baselines: **DISC-R** (random step selection), **DISC-negQ**, and **DISC-negZ** (which prioritize the opposite steps of DISC-Q and DISC-Z, respectively). As shown in Fig. 9, the selection of a priority metric significantly impacts performance. Both DISC-Q and DISC-Z *significantly* outperform random selection and their inverse counterparts, demonstrating the effectiveness of their priority heuristics. Additional details and analysis are in App. D.2.

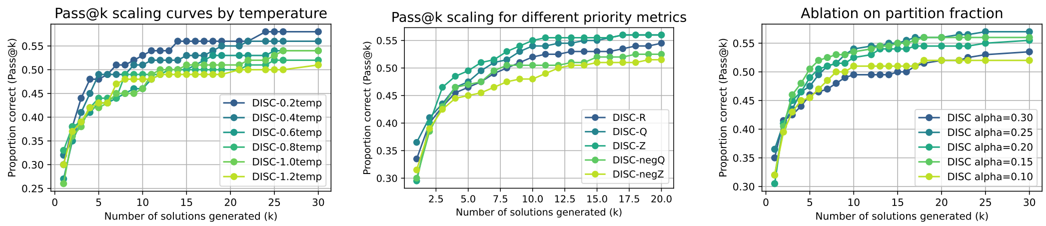

Figure 9: **Analysis of factors affecting DISC performance on APPS with gpt-4o-mini.** (Left) Effect of temperature: Unlike BoN and other inference scaling methods, DISC achieves higher performance at lower temperatures. (Middle) Effect of priority metrics: Both Q-based and Z-based priority metrics outperform random selection and their inverses, highlighting their effectiveness. (Right) Effect of partition fraction α: The range $0.15 \leq \alpha \leq 0.25$ appears optimal.

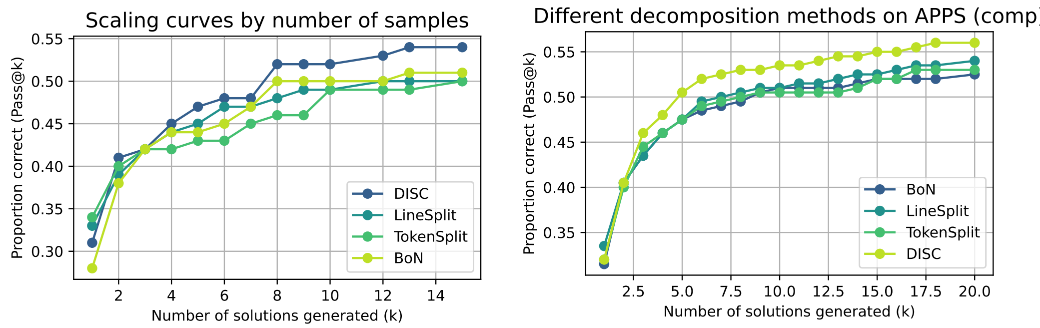

Figure 10: **Comparison of Pass@k performance on APPS with ground truth tests (right) and self generated validation tests (left) using gpt-4o-mini.** DISC scales more effectively in both.

4.7 Ablation on Base LLM Model

We evaluate DISC across different LLMs, including open-source models. As shown in Fig. 8 and Fig. 21, DISC significantly enhances performance even for weaker models. Specifically, it improves Llama's pass rate from 1% to 5.5%, a *550% relative increase*, and Mistral's from 0% to 3.5%, demonstrating substantial gains even from a nonzero baseline. Additional details and analyses are in App. D.3.

4.8 Ablation on Partition Fraction α

We conduct an ablation study to analyze the effect of the partition fraction α on DISC performance. As shown in Fig. 9 and 25, the optimal range appears to be $0.15 \leq \alpha \leq 0.25$. Lower partition fractions ($\alpha < 0.5$) tend to perform better due to the *asymmetric cost of sampling from different halves of the partition*. Sampling from the first half requires generating more tokens, while the second half requires fewer, making it crucial to partition the first half more conservatively. Additional analysis are in App. D.4.

4.9 Search and DISC

We demonstrate that search methods such as **MCTS** and **beam search** can be combined with DISC. As shown in Fig. 41 in the Appendix, greedy search explores deeper partitions given the same search budget due to its greedy nature, while MCTS and beam search reach similar, shallower depths. However, MCTS allocates the search budget more effectively than beam search, leading to higher performance, as seen in Fig. 8. Additional details and analysis are in App. F.

5 Conclusion

We introduce DISC, a dynamic decomposition framework that adaptively partitions solution steps based on difficulty, improving inference scaling by directing compute toward critical steps while balancing exploration and resource allocation. DISC seamlessly integrates with search-based methods such as MCTS and beam search, further enhancing performance. It also identifies challenging steps for LLMs, aiding curriculum learning, fine-tuning, and dataset augmentation. By dynamically adjusting partitioning based on available compute, DISC enables more adaptive and efficient reasoning in large language models, with broad implications for both training and inference optimization.

REFERENCES

Arash Ahmadian, Chris Cremer, Matthias Gallé, Marzieh Fadaee, Julia Kreutzer, Olivier Pietquin, Ahmet Üstün, and Sara Hooker. Back to basics: Revisiting reinforce style optimization for learning from human feedback in llms. *arXiv preprint arXiv:2402.14740*, 2024.

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

Bradley Brown, Jordan Juravsky, Ryan Ehrlich, Ronald Clark, Quoc V Le, Christopher Ré, and Azalia Mirhoseini. Large language monkeys: Scaling inference compute with repeated sampling. *arXiv preprint arXiv:2407.21787*, 2024.

Alexandra Carpentier and Michal Valko. Extreme bandits. *Advances in Neural Information Processing Systems*, 27, 2014.

Tristan Cazenave. Nested monte-carlo search. In *Twenty-First International Joint Conference on Artificial Intelligence*, 2009.

Angelica Chen, David Dohan, and David So. Evoprompting: Language models for code-level neural architecture search. *Advances in neural information processing systems*, 36:7787–7817, 2023a.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. Codet: Code generation with generated tests. *arXiv preprint arXiv:2207.10397*, 2022.

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

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

Xingyu Chen, Jiahao Xu, Tian Liang, Zhiwei He, Jianhui Pang, Dian Yu, Linfeng Song, Qiuzhi Liu, Mengfei Zhou, Zhuosheng Zhang, Rui Wang, Zhaopeng Tu, Haitao Mi, and Dong Yu. Do not think that much for 2+3=? on the overthinking of o1-like llms, 2024b. URL https://arxiv.org/abs/2412.21187.

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

Vincent A Cicirello and Stephen F Smith. The max k-armed bandit: A new model of exploration applied to search heuristic selection. In *The Proceedings of the Twentieth National Conference on Artificial Intelligence*, volume 3, pp. 1355–1361, 2005.

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

Dheeru Dua, Shivanshu Gupta, Sameer Singh, and Matt Gardner. Successive prompting for decomposing complex questions. *arXiv preprint arXiv:2212.04092*, 2022.

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

Kanishk Gandhi, Denise Lee, Gabriel Grand, Muxin Liu, Winson Cheng, Archit Sharma, and Noah D Goodman. Stream of search (sos): Learning to search in language. *arXiv preprint arXiv:2404.03683*, 2024.

Xinyu Guan, Li Lyna Zhang, Yifei Liu, Ning Shang, Youran Sun, Yi Zhu, Fan Yang, and Mao Yang. rstar-math: Small llms can master math reasoning with self-evolved deep thinking, 2025. URL https://arxiv.org/abs/2501.04519.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

Shibo Hao, Yi Gu, Haodi Ma, Joshua Hong, Zhen Wang, Daisy Wang, and Zhiting Hu. Reasoning with language model is planning with world model. In *Empirical Methods in Natural Language Processing*, pp. 8154–8173, 2023.

Erik Hemberg, Stephen Moskal, and Una-May O'Reilly. Evolving code with a large language model. *Genetic Programming and Evolvable Machines*, 25(2):21, 2024.

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

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

Sergio Hernández-Gutiérrez, Minttu Alakuijala, Alexander V Nikitin, and Pekka Marttinen. Recursive decomposition with dependencies for generic divide-and-conquer reasoning. In *The First Workshop on System-2 Reasoning at Scale, NeurIPS'24*, 2024.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*, 2024.

Tushar Khot, Harsh Trivedi, Matthew Finlayson, Yao Fu, Kyle Richardson, Peter Clark, and Ashish Sabharwal. Decomposed prompting: A modular approach for solving complex tasks. *arXiv preprint arXiv:2210.02406*, 2022.

Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large language models are zero-shot reasoners. *Advances in neural information processing systems*, 35: 22199–22213, 2022.

Kuang-Huei Lee, Ian Fischer, Yueh-Hua Wu, Dave Marwood, Shumeet Baluja, Dale Schuurmans, and Xinyun Chen. Evolving deeper llm thinking, 2025. URL https://arxiv.org/abs/2501.09891.

Kuang-Huei Leea, Ian Fischera, Yueh-Hua Wuc, Dave Marwood, Shumeet Baluja, Dale Schuurmans, and Xinyun Chen. Evolving deeper llm thinking. *Gen*, 2:3, 2025.

Joel Lehman, Jonathan Gordon, Shawn Jain, Kamal Ndousse, Cathy Yeh, and Kenneth O Stanley. Evolution through large models. In *Handbook of Evolutionary Machine Learning*, pp. 331–366. Springer, 2023.

Kyla H Levin, Kyle Gwilt, Emery D Berger, and Stephen N Freund. Effective llm-driven code generation with pythoness. *arXiv preprint arXiv:2501.02138*, 2025.

Yifei Li, Zeqi Lin, Shizhuo Zhang, Qiang Fu, Bei Chen, Jian-Guang Lou, and Weizhu Chen. Making language models better reasoners with step-aware verifier. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 5315–5333, 2023.

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

Jonathan Light, Min Cai, Weiqin Chen, Guanzhi Wang, Xiusi Chen, Wei Cheng, Yisong Yue, and Ziniu Hu. Strategist: Learning strategic skills by llms via bi-level tree search. *arXiv preprint arXiv:2408.10635*, 2024a.

Jonathan Light, Yue Wu, Yiyou Sun, Wenchao Yu, Xujiang Zhao, Ziniu Hu, Haifeng Chen, Wei Cheng, et al. Scattered forest search: Smarter code space exploration with llms. *arXiv preprint arXiv:2411.05010*, 2024b.

Hunter Lightman, Vineet Kosaraju, Yura Burda, Harri Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step. *arXiv preprint arXiv:2305.20050*, 2023.

Zicheng Lin, Tian Liang, Jiahao Xu, Qiuzhi Lin, Xing Wang, Ruilin Luo, Chufan Shi, Siheng Li, Yujiu Yang, and Zhaopeng Tu. Critical tokens matter: Token-level contrastive estimation enhances llm's reasoning capability, 2025. URL https://arxiv.org/abs/2411.19943.

Vadim Liventsev, Anastasiia Grishina, Aki Härmä, and Leon Moonen. Fully autonomous programming with large language models. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pp. 1146–1155, 2023.

Rohin Manvi, Anikait Singh, and Stefano Ermon. Adaptive inference-time compute: Llms can predict if they can do better, even mid-generation. *arXiv preprint arXiv:2410.02725*, 2024.

Harsha Nori, Naoto Usuyama, Nicholas King, Scott Mayer McKinney, Xavier Fernandes, Sheng Zhang, and Eric Horvitz. From medprompt to o1: Exploration of run-time strategies for medical challenge problems and beyond, 2024. URL https://arxiv.org/abs/2411.03590.

Giambattista Parascandolo, Lars Buesing, Josh Merel, Leonard Hasenclever, John Aslanides, Jessica B Hamrick, Nicolas Heess, Alexander Neitz, and Theophane Weber. Divide-and-conquer monte carlo tree search for goal-directed planning. *arXiv preprint arXiv:2004.11410*, 2020.

Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D Manning, Stefano Ermon, and Chelsea Finn. Direct preference optimization: Your language model is secretly a reward model. *Advances in Neural Information Processing Systems*, 36, 2024.

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M Pawan Kumar, Emilien Dupont, Francisco JR Ruiz, Jordan S Ellenberg, Pengming Wang, Omar Fawzi, et al. Mathematical discoveries from program search with large language models. *Nature*, 625(7995):468–475, 2024.

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

Zayne Sprague, Fangcong Yin, Juan Diego Rodriguez, Dongwei Jiang, Manya Wadhwa, Prasann Singhal, Xinyu Zhao, Xi Ye, Kyle Mahowald, and Greg Durrett. To cot or not to cot? chain-of-thought helps mainly on math and symbolic reasoning, 2024. URL https://arxiv.org/abs/2409.12183.

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

Evan Wang, Federico Cassano, Catherine Wu, Yunfeng Bai, Will Song, Vaskar Nath, Ziwen Han, Sean Hendryx, Summer Yue, and Hugh Zhang. Planning in natural language improves LLM search for code generation. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=48WAZhwHHw.

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

Xuezhi Wang and Denny Zhou. Chain-of-thought reasoning without prompting, 2024. URL https://arxiv.org/abs/2402.10200.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. In *The Eleventh International Conference on Learning Representations*, 2023a. URL https://openreview.net/forum?id=1PL1NIMMrw.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V Le, Ed H Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. In *International Conference on Learning Representations*, 2023b.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *NeurIPS*, 35: 24824–24837, 2022.

Yangzhen Wu, Zhiqing Sun, Shanda Li, Sean Welleck, and Yiming Yang. Inference scaling laws: An empirical analysis of compute-optimal inference for problem-solving with language models, 2024. URL https://arxiv.org/abs/2408.00724.

Yuxi Xie, Kenji Kawaguchi, Yiran Zhao, James Xu Zhao, Min-Yen Kan, Junxian He, and Michael Xie. Self-evaluation guided beam search for reasoning. *Advances in Neural Information Processing Systems*, 36, 2024.

Wei Xiong, Hanning Zhang, Nan Jiang, and Tong Zhang. An implementation of generative prm. https://github.com/RLHFlow/RLHF-Reward-Modeling, 2024.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. *Advances in Neural Information Processing Systems*, 36, 2024.

Eric Zelikman, Qian Huang, Gabriel Poesia, Noah Goodman, and Nick Haber. Parsel: Algorithmic reasoning with language models by composing decompositions. *Advances in Neural Information Processing Systems*, 36:31466–31523, 2023.

Zhiyuan Zeng, Qinyuan Cheng, Zhangyue Yin, Bo Wang, Shimin Li, Yunhua Zhou, Qipeng Guo, Xuanjing Huang, and Xipeng Qiu. Scaling of search and learning: A roadmap to reproduce o1 from reinforcement learning perspective, 2024. URL https://arxiv.org/abs/2412.14135.

Janis Zenkner, Lukas Dierkes, Tobias Sesterhenn, and Chrisitan Bartelt. Abstractbeam: Enhancing bottom-up program synthesis using library learning. *arXiv preprint arXiv:2405.17514*, 2024.

Lunjun Zhang, Arian Hosseini, Hritik Bansal, Mehran Kazemi, Aviral Kumar, and Rishabh Agarwal. Generative verifiers: Reward modeling as next-token prediction, 2024. URL https://arxiv.org/abs/2408.15240.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. Judging llm-as-a-judge with mt-bench and chatbot arena, 2023. URL https://arxiv.org/abs/2306.05685.

Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. Language agent tree search unifies reasoning acting and planning in language models. *ICML*, 2024.

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

Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Claire Cui, Olivier Bousquet, Quoc V Le, et al. Least-to-most prompting enables complex reasoning in large language models. In *International Conference on Learning Representations*, 2023.

A Code implementation of DISC

Python implementation of DISC

```python
def dynamic_decomposition(problem, model, reward_model, split_str, complete_solution, fraction,
      solution_budget, split_metric, stop_threshold=-float("inf"), stop_sum_score=1.0, stop_if_solved=
      False, ):
    """
    Decomposes the solution using a dynamic binary search approach

    Args:
        problem (Problem): The problem to solve
        model (Model): The model to use for generation
        reward_model (function): The reward model to use for scoring
        split_str (function): The function to use for splitting a string
        complete_solution (function): The function to use for completing a solution
        fraction (float): The fraction to split the string
        solution_budget (int): The maximum number of solutions to generate
        split_metric (function): The metric to use for splitting
        stop_threshold (float): The threshold to stop splitting
        stop_sum_score (float): The sum score to stop generating completions
        stop_if_solved (bool): Whether to stop if the problem is solved
    """

    # Initialize results and decomposition steps
    decomp_return = {
        "generated_solutions": [],
        "decomposition": []
    }

    while len(decomp_return["generated_solutions"]) < solution_budget:
        # Combine all previous steps into an intermediate solution
        intermediate_solution = "".join([step["step_str"] for step in decomp_return["decomposition"]])
        new_scores = []
        best_solution = None
        best_completion = None
        best_score = -float("inf")
        sum_score = 0.0

        # 1) Generate completions until we generate enough samples to estimate the split metric
        while sum_score < stop_sum_score:
            proposed_completion = complete_solution(problem, intermediate_solution, model)
            proposed_solution = intermediate_solution + proposed_completion
            decomp_return["generated_solutions"].append(proposed_solution)

            # Update scores
            proposed_score = reward_model(proposed_solution)
            new_scores.append(proposed_score)
            sum_score += proposed_score

            # Track the best solution
            if proposed_score > best_score:
                best_solution = proposed_solution
                best_score = proposed_score
                best_completion = proposed_completion

            # Stop early if problem is solved
            if stop_if_solved and proposed_score >= 1.0:
                decomp_return["decomposition"].append({"step_str": proposed_completion})
                return decomp_return

        new_metric = split_metric(new_scores)
        last_metric = decomp_return["decomposition"][-1]["metric"] if decomp_return["decomposition"]
             else None

        # Determine the split target. We always split the step with the highest metric
        is_split_new_step = last_metric is None or new_metric >= last_metric
        split_target = decomp_return["decomposition"][-1]["step_str"] if not is_split_new_step else
             best_completion

        # 3) Attempt to split the target
        split_result = split_str(split_target, fraction)
        if not split_result: # If we can't split the target, we're done
            decomp_return["decomposition"].append({"step_str": best_completion, "metric": new_metric})
            return decomp_return

        # Update decomposition based on split
        part1, part2 = split_result
        if is_split_new_step:
            decomp_return["decomposition"].append({"step_str": part1, "metric": new_metric})
            # Stopping condition based on threshold
            if new_metric < stop_threshold:
                decomp_return["decomposition"].append({"step_str": part2})
                return decomp_return
        else:
            decomp_return["decomposition"][-1] = {"step_str": part1, "metric": last_metric}

    return decomp_return
```

B PSEUDOCODE FOR DISC

Algorithm 1 Dynamic Decomposition

Input: Problem instance x, reward model r, partition function f, LLM policy model π, partition fraction α, solution budget B, priority metric h, metric stopping precision θ, sampling stopping threshold σ, is inference mode $\mathbf{b}_{\text{inference}}$ **Output:** Final decomposition D Initialize $D \leftarrow \{\text{generated\_solutions} : \varnothing, \text{decomposition} : \varnothing\}$ # Decompose the solution recursively until we reach the desired precision θ or run out of budget B $|D.\text{generated\_solutions}| < B$ $y_{\text{intermediate}} \leftarrow \text{Concatenate}([\text{step.step\_str} \; \forall \; \text{step} \in D.\text{decomposition}])$ ▷ Concatenate previous steps to form intermediate solution $R_{\text{new}} \leftarrow \varnothing$ ▷ Record rewards of completions best.$y_{\text{final}} \leftarrow$ None, best.$y_{\text{completion}} \leftarrow$ None, best.$r \leftarrow -\infty$ ▷ Track the best completion # Step 1: Generate completions until we have enough samples to estimate the splitting metric. Here we use a geometric sampling distribution $\text{sum}(R_{\text{new}}) < \sigma$ $y_{\text{completion}} \leftarrow \pi(\cdot|x, y_{\text{intermediate}})$ $y_{\text{proposed}} \leftarrow y_{\text{intermediate}} \oplus y_{\text{completion}}$ Append y_{proposed} to D.generated_solutions $r_{\text{proposed}} \leftarrow r(y_{\text{proposed}})$ Append r_{proposed} to R_{new} $r_{\text{proposed}} >$ best.r best.$y_{\text{final}} \leftarrow y_{\text{proposed}}$, best.$y_{\text{completion}} \leftarrow y_{\text{completion}}$ best.$r \leftarrow r_{\text{proposed}}$ $\mathbf{b}_{\text{inference}}$ **and** $r_{\text{proposed}} = 1.0$ Append $\{\text{step\_str} : y_{\text{completion}}\}$ to D.decomposition **Return** D ▷ Exit if problem is solved # Step 2: Compute splitting metric $z_{\text{new}} \leftarrow h(R_{\text{new}})$ $z_{\text{last}} \leftarrow D.\text{decomposition}[-1].z$ **if** D.decomposition $\neq \varnothing$ **else** $-\infty$ # Step 3: Split the step with the higher metric $\mathbf{b}_{\text{split new step}} \leftarrow z_{\text{new}} \geq z_{\text{last}}$ $y_{\text{target step}} \leftarrow$ best.$y_{\text{completion}}$ **if** $\mathbf{b}_{\text{split new step}}$ **else** $D.\text{decomposition}[-1].\text{step\_str}$ $y_1, y_2 \leftarrow f(y_{\text{target step}}, \alpha)$ $y_1 =$ None **or** $y_2 =$ None Append $\{\text{step\_str} : y_{\text{completion}}, \text{metric} : z_{\text{new}}\}$ to D.decomposition **Return** D ▷ Exit if we cannot do a finer split $\mathbf{b}_{\text{split new step}}$ Append $\{\text{step\_str} : y_1, \text{metric} : z_{\text{new}}\}$ to D.decomposition ▷ Add new step $z_{\text{new}} < \theta$ Append $\{\text{step\_str} : y_2\}$ to D.decomposition **Return** D ▷ Exit if all metrics are smaller than precision $D.\text{decomposition}[-1] \leftarrow \{\text{step\_str} : y_1, \text{metric} : z_{\text{last}}\}$ ▷ Split last step **Return** D

C RELATED WORK

Inference scaling. Inference scaling has emerged as a dominant paradigm, driven by the introduction of o1- and r1-like chain-of-thought reasoning models (Snell et al., 2024; Brown et al., 2024; Manvi et al., 2024; Leea et al., 2025). Several works examine the trade-off between inference compute and training compute (Guan et al., 2025; Chen et al., 2024b). LLM inference often relies on decomposing complex problems into intermediate reasoning steps, as seen in chain-of-thought (CoT) prompting (Wei et al., 2022; Sprague et al., 2024; Wang & Zhou, 2024) and its variants (Kojima et al., 2022; Zhou et al., 2023; Wang et al., 2023b; Li et al., 2023). We extend inference scaling by introducing a new approach for adaptive compute allocation (Manvi et al., 2024).

LLM reasoning and code generation. LLM reasoning and code generation are central tasks for inference scaling. Evolutionary inference scaling methods have been explored in program generation (Liventsev et al., 2023; Chen et al., 2023a; Romera-Paredes et al., 2024; Lehman et al., 2023; Hemberg et al., 2024). Domain-specific decomposition strategies have been applied in code generation, such as function-based decomposition (Chen et al., 2024a; Zenkner et al., 2024; Levin et al., 2025). More broadly, decomposition often involves prompting LLMs to generate subtask completions (Hernández-Gutiérrez et al., 2024; Khot et al., 2022; Dua et al., 2022), which differs from methods that refine a single LLM generation.

Reinforcement learning and Monte Carlo methods. Unlike standard RL, our setting resembles a search problem where the goal is to identify the single highest-reward path. Cazenave (2009) demonstrated that nested Monte Carlo search can accelerate optimal pathfinding. Under the bandit setting, this can be formulated as identifying the arm with the highest *maximum* reward rather than the highest mean reward (Cicirello & Smith, 2005; Carpentier & Valko, 2014).

D ABLATION STUDIES

D.1 ABLATION ON TEMPERATURE

We conduct an ablation study to analyze the effects of temperature on DISC and BoN. Temperature controls the randomness of token sampling in autoregressive models, influencing both exploration and consistency. Higher temperatures encourage more diverse outputs, whereas lower temperatures yield more deterministic generations. To examine its impact, we evaluate DISC and BoN on a 100-problem subset of APPS (the first 100 problems) using gpt-4o-mini.

Fig. 11 presents the Pass@token scaling curve for DISC across different temperatures. The results indicate that lower temperatures lead to improved performance, as DISC benefits from more deterministic step selection. Unlike BoN, which relies on broad solution sampling, DISC dynamically refines steps, making stable token probabilities advantageous.

Fig. 12 illustrates the frequency of actual partitions made by DISC at different temperatures. As temperature increases, the number of partitions fluctuates more, suggesting that high temperature introduces instability in step selection. Lower temperatures provide more structured decomposition, reducing unnecessary subdivisions.

In Fig. 13, we visualize the mean reward per step. The trend shows a linear increase in reward as step number grows, demonstrating that deeper decomposition results in progressively better solutions. This reinforces that DISC effectively allocates computation towards refining difficult steps.

The mean standard deviation per step is shown in Fig. 14. Lower temperatures yield lower standard deviations, confirming that DISC benefits from reduced variability in sample quality. This consistency allows for more reliable prioritization of difficult steps, enhancing overall inference efficiency.

For comparison, Fig. 16 and Fig. 15 display Pass@token and Pass@k scaling curves for BoN across different temperatures. Unlike DISC, BoN achieves peak performance at a temperature around 0.6-0.8, balancing diversity and consistency. Higher temperatures increase exploration but degrade precision, while lower temperatures hinder sample diversity, reducing the probability of obtaining high-quality completions.

These findings highlight the fundamental difference between DISC and BoN: DISC benefits from lower variance and stable decomposition, while BoN relies on broader exploration facilitated by moderate temperature settings. As a result, optimal temperature settings differ significantly between these methods, with DISC favoring deterministic sampling and BoN requiring a balance between diversity and coherence.

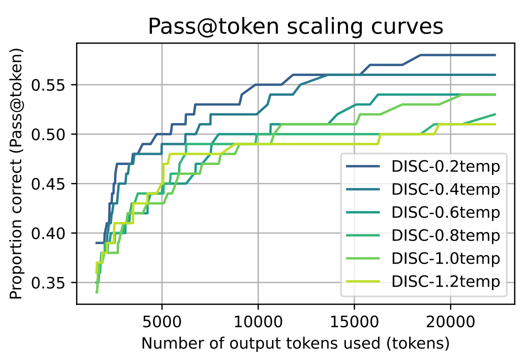

Figure 11: **Pass@token scaling curve for different temperatures on APPS using gpt-4o-mini**. The lower the temperature, the stronger the DISC performance.

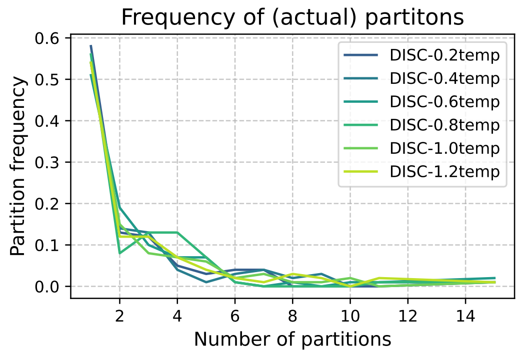

Figure 12: **Partition frequency of DISC with different temperatures on APPS using gpt-4o-mini**

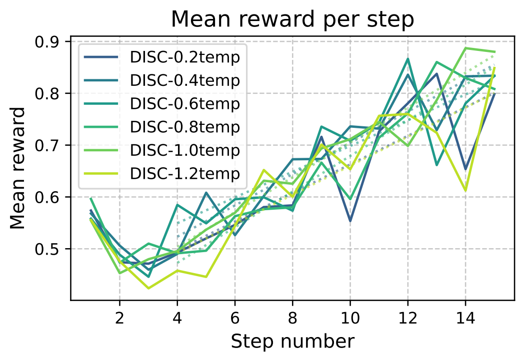

Figure 13: **Mean reward per step of DISC with different temperatures on APPS using gpt-4o-mini**. The mean reward scales linearly with step number.

D.2 Ablation on Priority Metric h

We analyze the effect of different priority metrics on DISC performance. We evaluate DISC using the first 200 competition-level APPS problems with gpt-4o-mini, setting the temperature to 0.8 for all experiments. The priority metric determines which steps are refined during recursive decomposition, impacting both efficiency and final solution quality.

Fig. 17 presents a token-level comparison of different priority metrics. Both DISC-Q and DISC-Z significantly outperform random selection and their inverse counterparts, demonstrating the importance of prioritizing high-value steps.

Fig. 18 illustrates the partition frequency under different priority metrics. We observe that effective metrics such as DISC-Q and DISC-Z lead to fewer, more meaningful partitions, whereas suboptimal strategies result in excessive, redundant partitioning.

The relationship between mean reward and step number is shown in Fig. 19. All tested metrics exhibit a strong correlation between increasing step depth and mean reward, indicating that decomposition progressively refines solutions. However, DISC-Q and DISC-Z achieve higher reward gains at earlier stages, suggesting that they prioritize the most impactful refinements.

Finally, Fig. 20 reports the standard deviation of rewards per step. Lower standard deviation suggests more stable solution quality, a property that DISC-Q and DISC-Z maintain better than random selection methods. This highlights their effectiveness in identifying and refining challenging steps efficiently.

Overall, these results confirm that choosing an appropriate priority metric is crucial for DISC. While DISC-Q and DISC-Z consistently enhance inference efficiency and quality, random or inverse strategies lead to poorer performance due to misallocation of compute resources.

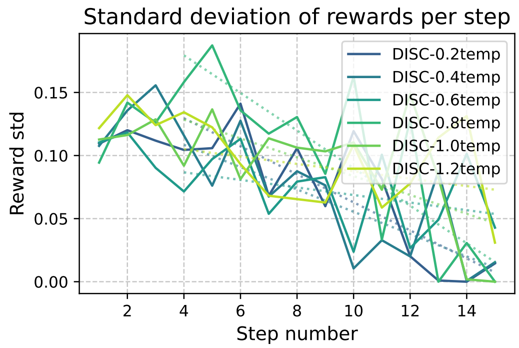

Figure 14: **Mean standard deviation per step of DISC with different temperatures on APPS using gpt-4o-mini**. Lower temperature means lower average standard deviation.

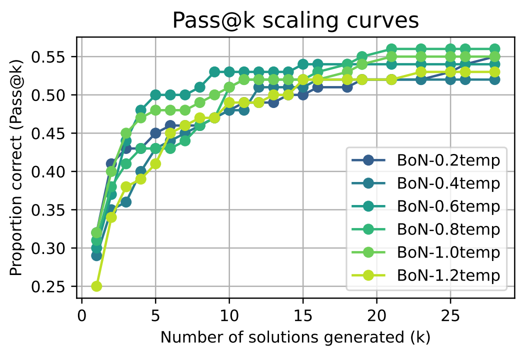

Figure 15: **Pass@k scaling curve for different temperatures on APPS using gpt-4o-mini for BoN**. A temperature around 0.6-0.8 leads to the best performance and balance between diversity and consistency.

D.3 Model Ablation

We investigate how different LLMs perform when used with DISC on 200 competition-level APPS problems, given a sample budget of 30. The groundtruth reward model was used to evaluate correctness, and all models were set to a temperature of 0.8. Due to the challenging nature of the benchmark, open-source models struggled to achieve strong performance independently. However, when paired with DISC, their performance significantly improved.

Figure 21 presents the Pass@token scaling curve for open-source models using DISC. The results demonstrate that DISC substantially enhances the capabilities of these models, closing the gap between them and proprietary alternatives.

Figure 22 visualizes the partition frequency of DISC with different open-source models. Compared to their standalone performance, the use of DISC led to more structured and effective decomposition, highlighting its adaptability to different architectures.

The mean reward per step is shown in Figure 23. Similar to prior findings, we observe that deeper decomposition leads to increasingly higher rewards. Notably, even lower-capacity models benefit from DISC 's ability to iteratively refine their solutions.

Finally, Figure 24 presents the mean standard deviation per step. With DISC, the variance in performance is significantly reduced, resulting in more stable and reliable inference.

Overall, these findings emphasize that DISC is a robust framework capable of enhancing inference performance across diverse LLMs, particularly those with limited standalone capabilities.

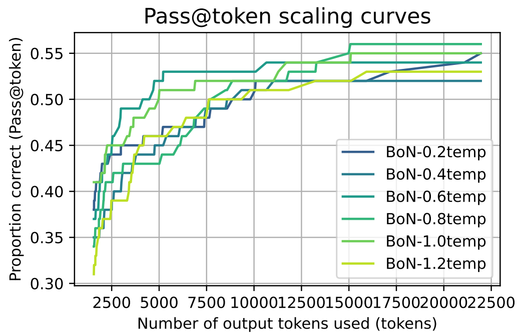

Figure 16: **Pass@token scaling curve for different temperatures on APPS using gpt-4o-mini for BoN**. A temperature around 0.6-0.8 leads to the best performance and balance between diversity and consistency.

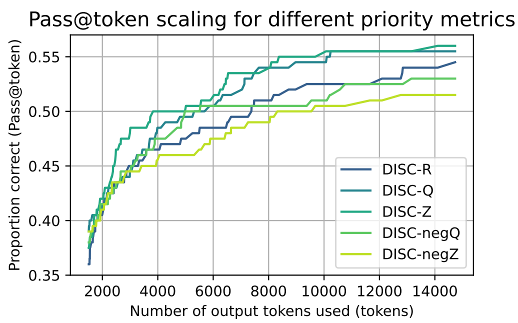

Figure 17: **Token level comparison of different priority metrics on DISC in the APPS setting with gpt-4o-mini.** Both Q and Z based priority metrics perform well.

D.4 Ablation on partition fraction α

We include some more analysis on the partition fraction here.

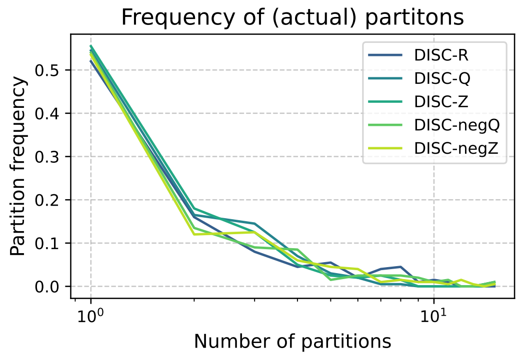

Figure 18: **Partition frequency of DISC with different priority metrics on APPS using gpt-4o-mini**

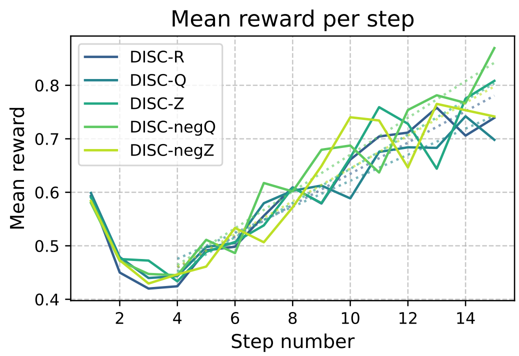

Figure 19: **Mean reward per step of DISC with different priority metrics on APPS using gpt-4o-mini**. All metrics display strong correlation between step depth and the mean reward.

E EXPERIMENTAL RESULTS EXTENDED

E.1 APPS

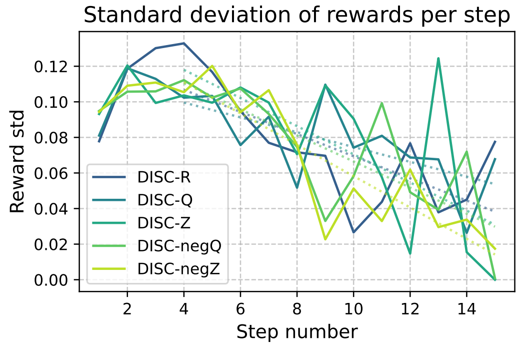

Figure 20: **Mean standard deviation per step of DISC with different priority metrics on APPS using gpt-4o-mini**. All metrics display correlation between step depth and the standard deviation.

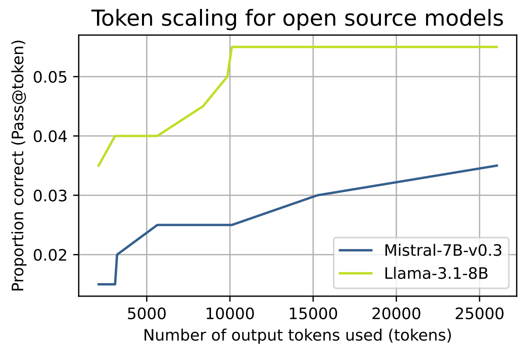

Figure 21: **Pass@token scaling curve for open source models with DISC on APPS**. DISC also demonstrates strong performance gains with open source models.

E.2 Additional examples

Below is another computed decomposition for the same problem as the one shown in the main text:

> Let x be the length and y be the width of the rectangle. Since the perimeter is 24 inches, we have $2x + 2y = 24 \Rightarrow x + y = 12$. Therefore, we must maximize the area xy subject to this constraint.
> We use the method of Lagrange multipliers. The Lagrangian is
>
> $$\mathcal{L}(x, y, \lambda) = xy - \lambda(x + y - 12).$$
>
> We differentiate with respect to x, y, and λ to obtain
>
> $$\frac{\partial \mathcal{L}}{\partial x} = y - \lambda = 0, \qquad \frac{\partial \mathcal{L}}{\partial y} = x - \lambda = 0, \qquad \frac{\partial \mathcal{L}}{\partial \lambda} = x + y - 12 = 0.$$
>
> We find that $x = y = \lambda$, so $x = y = \frac{12}{2} = 6$. Therefore, the maximum area of the rectangle is $6 \cdot 6 = \boxed{36}$.

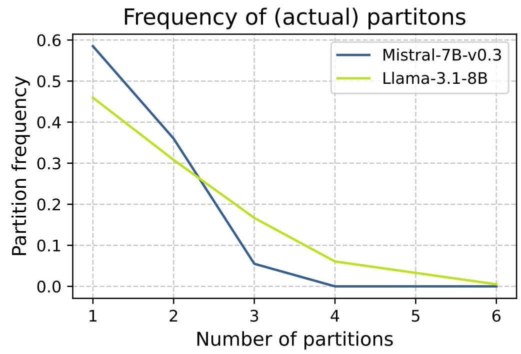

Figure 22: **Partition frequency of DISC with open source models on APPS**

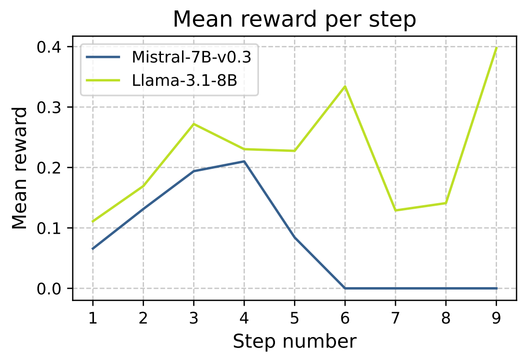

Figure 23: **Mean reward per step of DISC with open source models on APPS**

E.3 APPS WITH SELF-GENERATED VALIDATION TESTS

We examine DISC performance on APPS when using self-generated validation tests. All methods utilized the same set of self-generated validation tests to ensure fair comparisons. Each problem received 5-10 validation tests, with the exact number determined dynamically by the LLM. We evaluated a subset of 100 APPS problems, generating samples until the sample budget was exhausted or a correct solution was found.

Figure 27 illustrates the Pass@token scaling curve, showing that DISC maintains strong scaling performance in this setting, though at a slightly lower rate compared to ground-truth verification.

Figure 28 and Figure 29 compare actual and planned partition frequencies, respectively. The results indicate that DISC continues to make structured decompositions even with self-generated validation, preserving its efficiency.

The mean reward per step, shown in Figure 30, follows a similar trend as in previous experiments, reinforcing that DISC effectively allocates compute resources for iterative refinement.

Lastly, Figure 31 demonstrates that DISC maintains lower standard deviations in performance, indicating stable quality improvements across steps.

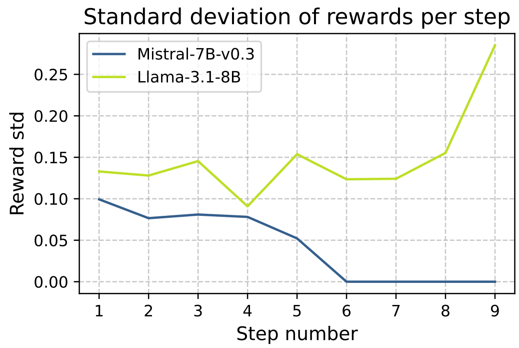

Figure 24: **Mean standard deviation per step of DISC with open source models on APPS**

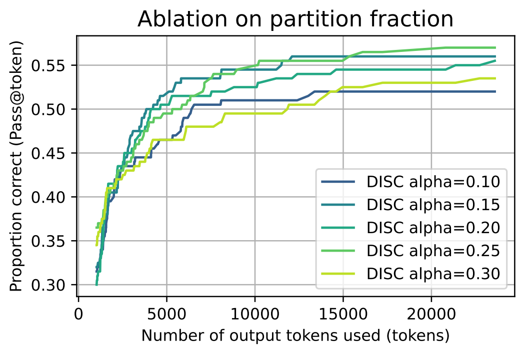

Figure 25: **Token level comparison of different DISC splitting fraction α on APPS competition level.** $0.15 \leq \alpha \leq 0.25$ seems to be optimal.

E.4 MATH500

Completions for MATH500 include both the reasoning steps and the final answer. Since MATH500 contains more problems than APPS200 and MATH problems tend to be relatively easier, solution quality saturates quickly. Therefore, we use a lower sample budget of 10 for these experiments.

Figure 32 presents the Pass@k performance for different decomposition methods on MATH500. We observe that all decomposition-based approaches achieve similar Pass@k performance, consistently outperforming BoN. This indicates that the structured nature of MATH problems allows multiple decomposition strategies to be effective.

Despite similar Pass@k results, the true advantage of DISC lies in its token efficiency, as shown in Figure 5. DISC significantly reduces the number of tokens required to reach correct solutions compared to alternative methods, demonstrating its ability to allocate computational effort efficiently in mathematical reasoning tasks.

Additionally, we analyze the partitioning behavior of DISC on MATH500. Figure 33 illustrates the actual partition frequency for different decomposition methods. The planned partitioning behavior, shown in Figure 34, further highlights how DISC effectively balances exploration and refinement.

Finally, we present the mean standard deviation per step in Figure 35. Lower variance suggests that DISC produces more stable and reliable decompositions over multiple runs, reinforcing its robustness in both mathematical and program synthesis domains.

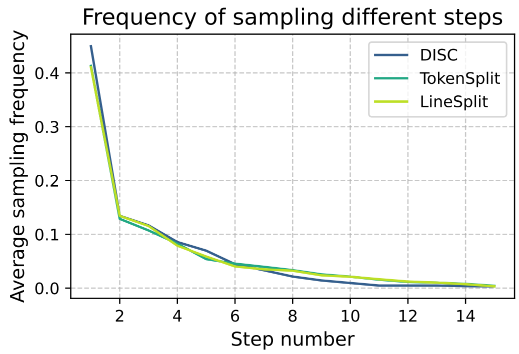

Figure 26: **Sampling frequency of each step averaged over the problems on APPS with gpt-4o-mini.** DISC seems to have a preferences for spending more compute on earlier found steps.

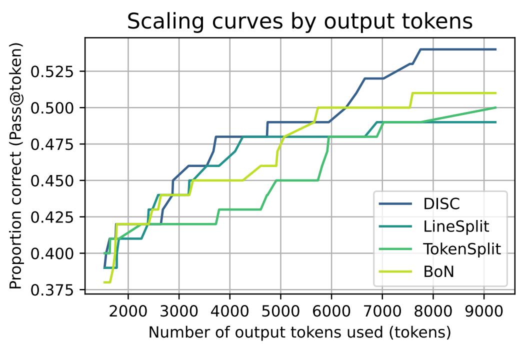

Figure 27: **Token level comparison of different decomposition methods on APPS with gpt-4o-mini and self-generated validation tests.** DISC still scales better than other methods in this setting, albeit at a lower rate.

E.5 LiveCodeBench

We evaluate DISC on LiveCodeBench, a benchmark designed for code generation tasks with a focus on real-world software development challenges. LiveCodeBench presents a unique set of problems requiring both reasoning and structured decomposition, making it a suitable testbed for evaluating DISC's ability to refine and improve intermediate steps.

Figure 36 shows the Pass@k comparison of different decomposition methods on LiveCodeBench. DISC consistently scales better than other decomposition methods, highlighting its ability to refine intermediate steps more effectively in complex coding scenarios.

Figure 37 illustrates the observed partition frequency of different decomposition methods. The structured approach of DISC results in well-balanced decomposition across steps, reducing unnecessary partitioning while maintaining sufficient granularity for improved solution refinement.

Figure 38 displays the planned partition frequency across methods. DISC dynamically determines the most effective partitions based on the evolving problem state, leading to more targeted and efficient decompositions.

Finally, Figure 39 presents the mean standard deviation per step across decomposition methods. Lower variance in DISC suggests that it produces more stable and reliable decompositions, reinforcing its robustness for solving LiveCodeBench problems.

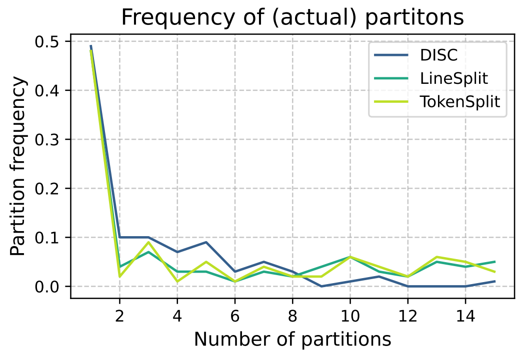

Figure 28: **Actual partition frequency of different decomposition methods on APPS with gpt-4o-mini and self-generated validation tests.**

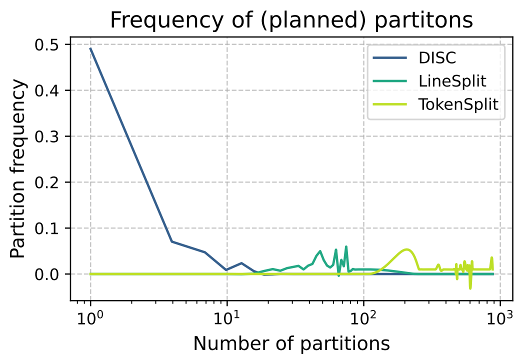

Figure 29: **Planned partition frequency of different decomposition methods on APPS with gpt-4o-mini and self-generated validation tests.**

F Search and scaling

F.1 Monte Carlo Tree Search (MCTS)

Monte Carlo Tree Search (MCTS) is a widely used algorithm for sequential decision-making in large search spaces, particularly in applications such as *game playing, planning, and inference scaling*. The algorithm builds a search tree incrementally by simulating different sequences of actions and updating estimates of state quality. A key advantage of MCTS is its ability to balance *exploration* (discovering new states) and *exploitation* (refining promising ones) using a data-driven search process. The MCTS pipeline consists of four fundamental steps: *selection, expansion, simulation, and backpropagation*.

F.1.1 Selection

Starting from the root node representing the current state $\boldsymbol{s}$, MCTS iteratively traverses the search tree by selecting child nodes based on a *selection policy*. The most commonly used selection criterion is the *Upper Confidence Bound for Trees (UCT)*, which balances exploration and exploitation:

$$UCT(\boldsymbol{s},\boldsymbol{d}) = \widehat{Q}(\boldsymbol{s},\boldsymbol{d}) + c\sqrt{\frac{\ln\left(\sum_{\boldsymbol{b}} n(\boldsymbol{s},\boldsymbol{b})\right)}{n(\boldsymbol{s},\boldsymbol{d})}}, \tag{1}$$

where $\widehat{Q}(\boldsymbol{s},\boldsymbol{d})$ represents the estimated value of selecting action $\boldsymbol{d}$ from state $\boldsymbol{s}$, $n(\boldsymbol{s},\boldsymbol{d})$ is the visit count for this action, and c is a hyperparameter controlling the trade-off between exploring new actions and favoring those with high past rewards.

Figure 30: **Mean reward per step of different decomposition methods on APPS with gpt-4o-mini and self-generated validation tests.**

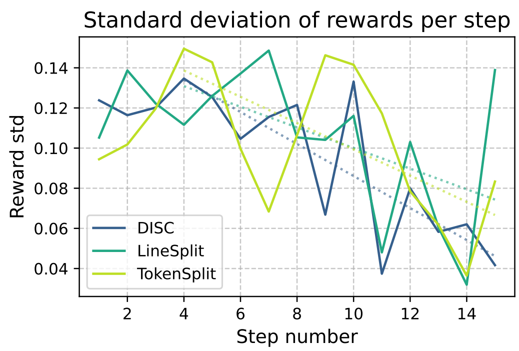

Figure 31: **Mean standard deviation different decomposition methods on APPS with gpt-4o-mini and self-generated validation tests.**

F.1.2 Expansion

Once a leaf node (a previously unexplored state) is reached, the algorithm expands the tree by *adding one or more new nodes*. These new nodes represent potential future states $\boldsymbol{s}'$ generated by sampling an action $\boldsymbol{d}$ from a predefined policy. This step broadens the search space and allows MCTS to evaluate new possibilities.

F.1.3 Simulation

Following expansion, the algorithm conducts a *simulation* (or rollout) from the newly added state. This step involves generating a sequence of actions according to a predefined policy until reaching a terminal state or an evaluation horizon. The outcome of the simulation, denoted as $v(\boldsymbol{s}')$, provides an estimate of the quality of the new state. Depending on the application, this could represent a *game result, an optimization score, or an inference accuracy metric*.

F.1.4 Backpropagation

The final step involves *propagating the results of the simulation back up the search tree* to refine the estimated values of prior states and actions. Each node along the trajectory $\tau = [\boldsymbol{s}_0, \boldsymbol{d}_1, \boldsymbol{s}_2, \ldots, \boldsymbol{s}_{-1}]$ is updated iteratively:

$$\widehat{Q}(\boldsymbol{s}_i, \boldsymbol{d}_{i+1})^{(t+1)} \leftarrow (1-\alpha_n)\widehat{Q}(\boldsymbol{s}_i, \boldsymbol{d}_{i+1})^{(t)} + \alpha_n \max\{\widehat{Q}(\boldsymbol{s}_i, \boldsymbol{d}_{i+1})^{(t)}, \widehat{Q}(\boldsymbol{s}_{i+1}, \boldsymbol{d}_{i+2})^{(t+1)}\}, \quad (2)$$

where α_n is a learning rate that depends on the visit count, and the maximum function ensures that the best-performing trajectories are emphasized.

MCTS has been widely adopted in inference scaling techniques due to its ability to *efficiently allocate computational resources*, focusing more on *high-reward states* while avoiding unnecessary

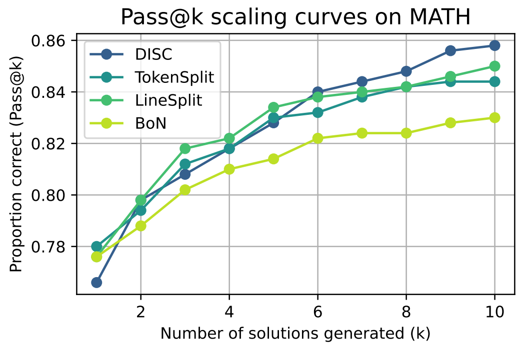

Figure 32: **Pass@k performance comparison for different decomposition methods on MATH500. DISC consistently outperforms BoN across different sampling budgets.**

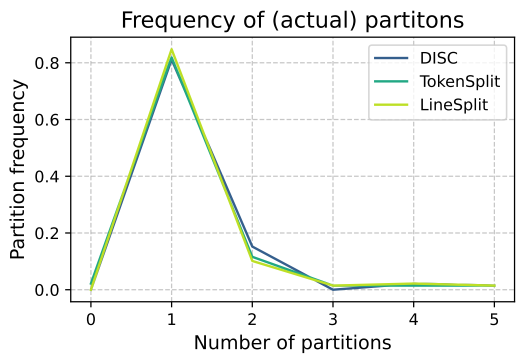

Figure 33: **Observed partition frequency of different decomposition methods on MATH500. DISC effectively segments problems into meaningful subcomponents.**

exploration of unpromising regions. In later sections, we explore how MCTS can be combined with *dynamic decomposition* to further optimize inference scaling.

F.1.5 Combining Dynamic Decomposition with MCTS

MCTS can be enhanced by integrating *dynamic decomposition*, where each node in the search tree represents a decomposition of the problem into steps. Instead of treating states as atomic decisions, we recursively decompose reasoning steps, dynamically adjusting granularity based on difficulty.

In this framework:

- Each node in the MCTS tree represents a partial decomposition of the problem, with child nodes corresponding to alternative step partitions.
- Branching occurs by generating candidate next steps using dynamic decomposition, allowing finer steps for complex regions while maintaining efficiency for simpler ones.
- The selection step prioritizes nodes that represent more promising decompositions, dynamically refining challenging areas through recursive subdivision.
- The backpropagation step ensures that decompositions leading to high-quality solutions are reinforced, helping the search tree converge toward optimal inference paths.

By integrating dynamic decomposition with MCTS, we efficiently allocate compute to the most critical reasoning steps, improving inference quality while maintaining computational efficiency.

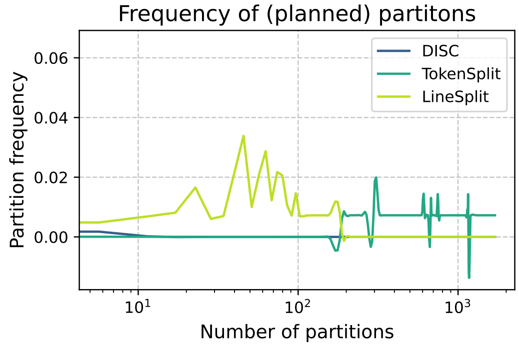

Figure 34: **Planned partitioning strategy of different decomposition methods on MATH500. DISC 's structured approach leads to more efficient problem breakdowns.**

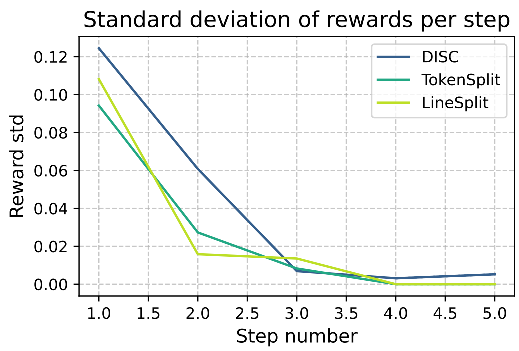

Figure 35: **Mean standard deviation per step for different decomposition methods on MATH500. Lower variance in DISC suggests more stable and reliable problem-solving steps.**

F.2 BEAM SEARCH

Beam search is a heuristic search algorithm commonly used in inference tasks where computational efficiency is a priority. Unlike exhaustive search methods, beam search maintains only the top k best candidates at each step, making it an effective strategy for structured prediction problems and sequential decision-making.

At each iteration:

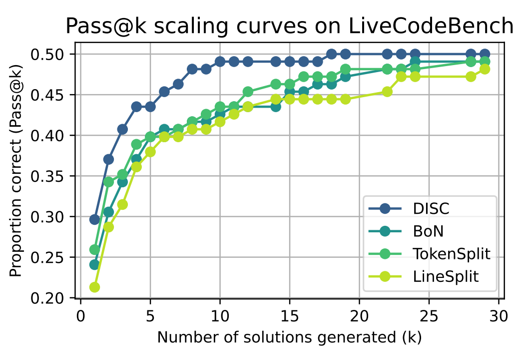

Figure 36: **Pass@k performance comparison for different decomposition methods on LiveCodeBench.** DISC consistently outperforms other methods in structured problem refinement.

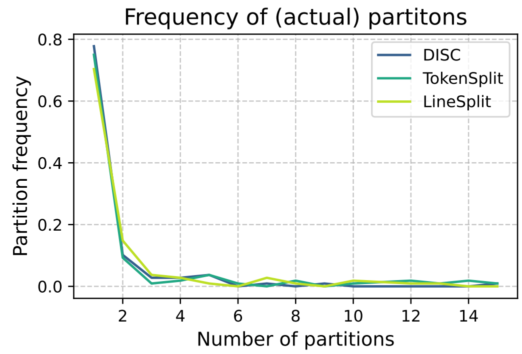

Figure 37: **Observed partition frequency of different decomposition methods on LiveCodeBench.** DISC effectively balances problem segmentation while avoiding excessive partitioning.

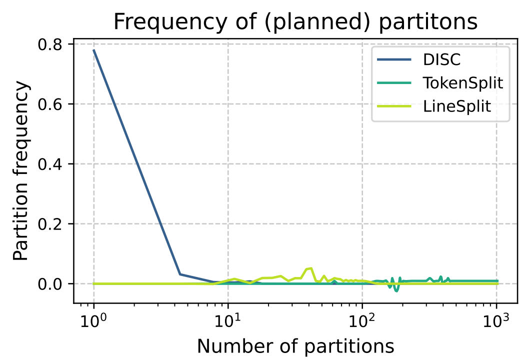

Figure 38: **Planned partitioning strategy of different decomposition methods on LiveCodeBench.** DISC dynamically adapts its partitioning to optimize search efficiency.

- The algorithm selects the k most promising partitions from the previous step based on an evaluation metric.
- Each selected partition is expanded by generating possible next-step samples.
- The newly generated partitions are ranked, and only the top k candidates are retained for the next iteration.
- This process continues until a stopping criterion is met, such as reaching a predefined depth or finding a sufficiently high-quality solution.

Beam search provides a computationally efficient way to explore structured solution spaces while maintaining high-quality search trajectories. By integrating beam search with dynamic decomposition, we ensure that inference computation is allocated efficiently, focusing on the most promising reasoning paths at each step.

F.3 Additional Results and Analysis

Experiments comparing different search methods were conducted on a 100-problem subset of the APPS dataset (first 100 problems) using GPT-4o-mini. All methods used a temperature of 0.2, with $\alpha = 0.15$, Q priority metric, and $\sigma = 1.0$.

Token-level comparison: As shown in Figure 40, MCTS scales best among the tested methods, demonstrating superior efficiency in identifying promising partitions. Greedy search follows closely, while beam search exhibits the slowest scaling.

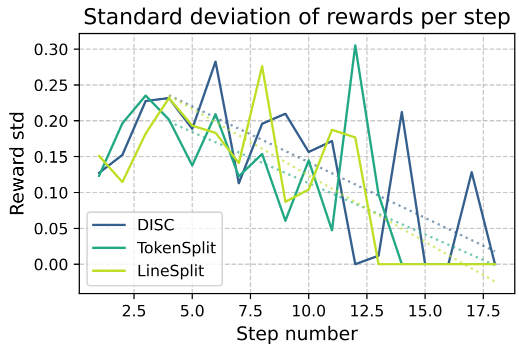

Figure 39: **Mean standard deviation per step for different decomposition methods on LiveCodeBench.** Lower variance in DISC suggests more stable and reliable problem-solving steps.

Partition frequency analysis: Figure 41 reveals that greedy search explores to greater depths within the same sampling budget. This suggests that greedy search prioritizes deep refinements, whereas MCTS and beam search balance depth with breadth.

Step variance analysis: Figure 42 illustrates that all search methods display decreasing standard deviation with increasing search depth. This trend indicates that deeper searches converge towards stable, high-quality partitions, reinforcing the benefits of dynamic decomposition.

These results highlight the trade-offs between search methods: MCTS offers robust exploration-exploitation balance, greedy search favors depth-first refinement, and beam search provides a structured yet computationally constrained approach. The integration of dynamic decomposition further enhances these search strategies by adaptively allocating computational resources to critical reasoning steps.

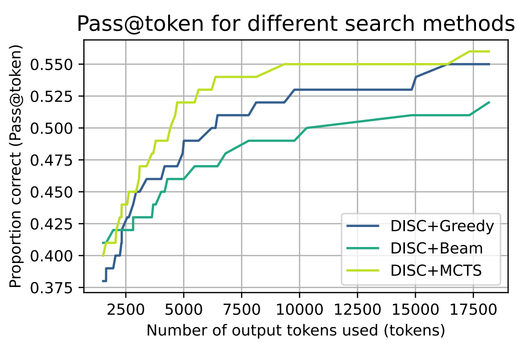

Figure 40: **Token level comparison of different decomposition search methods combined with DISC on APPS with gpt-4o-mini.** MCTS scales best, followed by greedy search, followed by beam search.

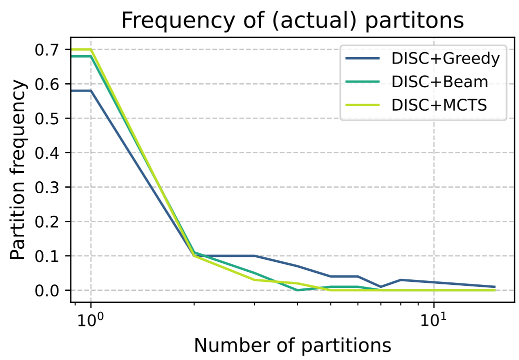

Figure 41: **Actual partition frequency of different decomposition search methods combined with DISC on APPS with gpt-4o-mini.** Greedy is able to search to higher depths given the same sampling budget.

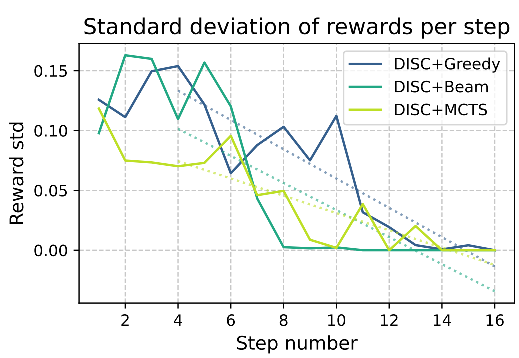

Figure 42: **Mean standard deviation of different decomposition search methods combined with DISC on APPS with gpt-4o-mini.** All search methods display decreasing standard deviation with search depth.