WIDER OR DEEPER? SCALING LLM INFERENCE-TIME COMPUTE WITH ADAPTIVE BRANCHING TREE SEARCH

Kou Misaki, Yuichi Inoue, Yuki Imajuku, So Kuroki, Taishi Nakamura, Takuya Akiba
Sakana AI, Japan
{kou.misaki, takiba}@sakana.ai

ABSTRACT

Recent advances demonstrate that increasing inference-time computation can significantly boost the reasoning capabilities of large language models (LLMs). Although repeated sampling (i.e., generating multiple candidate outputs) is a highly effective strategy, it falls short when external feedback is available to guide response selection and refinement. In this work, we propose *Adaptive Branching Monte Carlo Tree Search (AB-MCTS)*, a novel inference-time framework that unifies repeated sampling with principled multi-turn exploration and exploitation. At each node in the search tree, AB-MCTS dynamically decides whether to "go wider" by expanding new candidate responses or "go deeper" by revisiting existing ones based on external feedback signals. We evaluate our method on complex coding and engineering tasks using frontier API models. Empirical results show that AB-MCTS consistently outperforms both repeated sampling and standard MCTS, underscoring the importance of combining the response diversity of LLMs with multi-turn solution refinement for effective inference-time scaling.

1 INTRODUCTION

Recent work has begun to reveal that scaling inference-time computation can significantly boost the performance of large language models (LLMs) on complex tasks. Traditionally, LLM performance improvements have stemmed from training-time scaling—namely, increasing the size of training datasets and model parameters. By contrast, inference-time scaling attempts to harness the capacity of a fixed, pretrained LLM by allocating more computational resources at inference. This approach aims to enhance the LLM's reasoning and problem-solving abilities on-the-fly, without further training.

A prominent approach of inference-time scaling is *repeated sampling*, which encompasses methods such as best-of-n, majority voting, or self-consistency (Wang et al., 2023; Brown et al., 2024; Liang et al., 2024). In this approach, multiple candidate outputs are generated independently from the same prompt, and a final solution is chosen—often via simple heuristics. Repeated sampling has proven effective in challenging tasks such as coding competitions (Li et al., 2022) and the ARC Challenge (Greenblatt, 2024). This strategy leverages the *unbounded generative capacity* of LLMs—their ability to produce an effectively infinite range of diverse responses from a single prompt. The success of repeated sampling illustrates that harnessing this capacity is paramount to achieving effective inference-time scaling.

However, repeated sampling focuses exclusively on *exploration* and lacks an explicit mechanism for *exploitation*. In certain real-world scenarios, one can obtain external feedback on a candidate solution. For instance, in coding tasks, one can run tests to evaluate the correctness of generated programs and gather feedback on how to improve them (Jain et al., 2024). In such settings, it is natural to want to select promising solutions and refine them based on available feedback, which repeated sampling alone cannot accomplish effectively.

Although a handful of multi-turn methods for exploration and exploitation at inference time do exist, most of them were designed before the effectiveness of large-scale inference-time computation and the unbounded generative capacity was fully realized. Consequently, these methods often do not

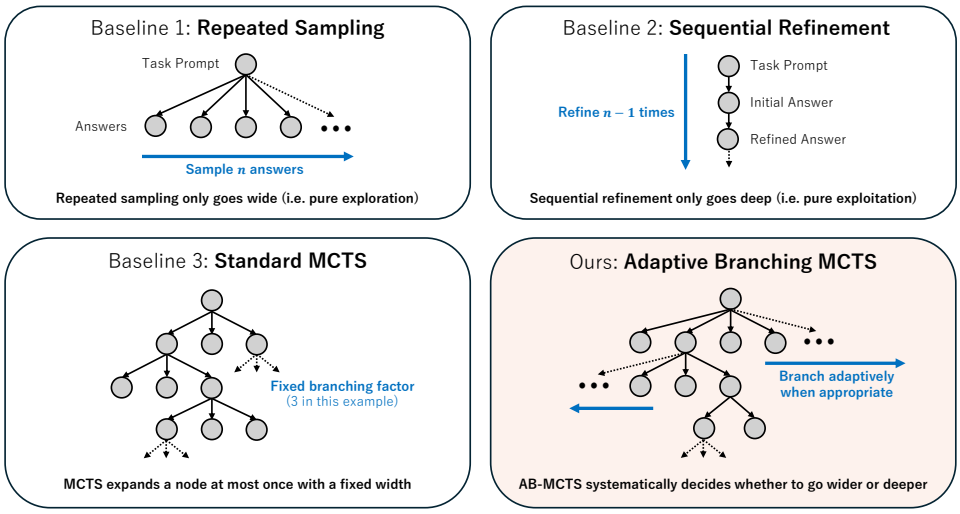

Figure 1: **A visual comparison of the proposed AB-MCTS against three baseline approaches.** Repeated sampling generates multiple answers at once (wide but shallow), sequential refinement iteratively revises a single solution (deep but narrow), and standard MCTS applies a fixed branching factor. In contrast, AB-MCTS dynamically decides whether to branch out or refine deeper in each iteration, effectively balancing exploration and exploitation.

fully leverage the unbounded generative capacity of LLMs. For example, standard Monte Carlo Tree Search (MCTS) has a fixed branching factor (i.e., the number of child nodes per state) as a hyperparameter (Zhou et al., 2023).

In this work, we propose *Adaptive Branching Monte Carlo Tree Search (AB-MCTS)*, a novel inference-time framework that unifies repeated sampling with multi-turn exploration and exploitation. Unlike traditional MCTS or beam search, AB-MCTS does not fix the width as a static hyperparameter. Instead, at each node of the search tree, AB-MCTS adaptively decides whether to "go wider" by generating new candidate responses or "go deeper" by refining existing ones, leveraging external feedback signals. This design naturally extends repeated sampling, allowing us to invoke the unbounded generative capacity of LLMs when necessary. Consequently, our framework provides a powerful mechanism for balancing exploration and exploitation in the context of LLM inference-time scaling.

We evaluated AB-MCTS on complex coding and engineering tasks using frontier API models such as GPT-4o and DeepSeek-V3, in an inference-time scaling scenario that allows up to 128 generation calls for each task instance. Under the same computational budget, AB-MCTS consistently achieved stronger results than previous approaches.

Our technical contributions are summarized as follows:

- **Unbounded Generative Capacity + Solution Refinement as a New Challenge.** We highlight the unbounded generative capacity of LLMs as key for inference-time scaling and propose the new task of combining it with solution refinement to manage the exploration–exploitation trade-off.
- **AB-MCTS Algorithm**. To address this challenge, we introduce AB-MCTS, which systematically decides whether to "go wide" or "go deep." We present two variants AB-MCTS-M and AB-MCTS-A based on different principles, each offering distinct trade-offs.
- **Empirical Validation on Frontier Models and Complex Tasks**. In a practical setting using frontier API models and real-world complex tasks, we show that AB-MCTS consistently outperforms existing methods.

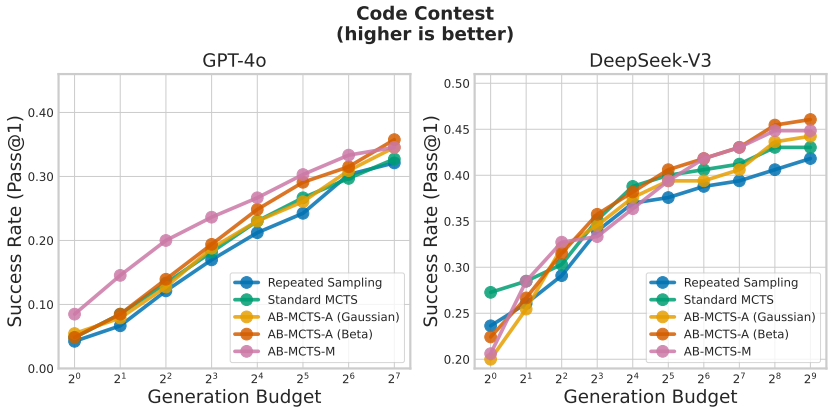

Figure 2: **Performance Comparison on Code Contest.** We compare five methods by plotting the success rate against the generation budget. The left plot shows GPT-4o's performance, and the right plot shows DeepSeek-V3's performance. The y-axis indicates the fraction of problems fully solved by the single best solution (Pass@1).

2 Proposed Method

Our inference-time search problem is defined as follows. We are given input, which may include task instructions or few-shot examples. The LLM can produce an answer to this input. Moreover, by providing the existing answer along with feedback on that answer (for instance, the results of executing code in a coding task) back to the LLM, we can refine the answer. Because the LLM is used with a non-zero temperature, both of these operations are stochastic, and the same input can yield different answers. A task-specific scoring evaluator is available. Our goal is to use these resources to arrive at a good answer that maximizes the score. See Appendix for details on the problem setting. Please see Appendix A.1 for further details on the problem setup and Appendix A.2 for how naive methods are expressed under this setting.

We consider constructing a tree T where each node N corresponds to an input text and its LLM-generated output. The expansion from the root does the direct answer generation, while the expansion from a node N employs the answer refinement using the input-output pairs of N's ancestors, including N itself. We formulate MCTS iteratively as follows. We begin with a single root node and expand the tree n_{nodes} times, ultimately producing $1 + n_{\text{nodes}}$ nodes in total. Each iteration proceeds in three steps:

1. **Selection:** choose a node N in the current tree for expansion.
2. **Expansion:** apply direct answer generation or answer refinement to N to create a new node N_{new}, appended as a child of N.
3. **Score backup:** propagate the score of N_{new} up the tree to update the ancestors' score information, including N_{new} itself.

Unlike the standard tasks typically tackled by MCTS where the number of possible actions at each node is finite, each call to an LLM can yield a new output even for the same input, making each node's branching factor theoretically infinite. To highlight this generative capacity, we introduce a "*GEN node*" that explicitly represents the action of generating a new child. GEN node exists as a child of all the nodes including the newly expanded ones, and represents the action of generating a new child and append it to the GEN node's parent node. When the GEN node is selected during selection, a new node is created at that location, i.e., the parent node is expanded. This means that not only leaf nodes but all nodes, including intermediate nodes, are candidates for expansion. This allows branching to occur adaptively, wherever and whenever it is needed. Further description of the framework can be found in Appendix A.3.

Here, the challenge lies in how to model the scoring probability of the nodes, including this new type of GEN node, during selection. For node selection, we adopt a Bayesian view of each action's score distribution and apply Thompson Sampling. In standard MCTS, the typical metric for choosing an

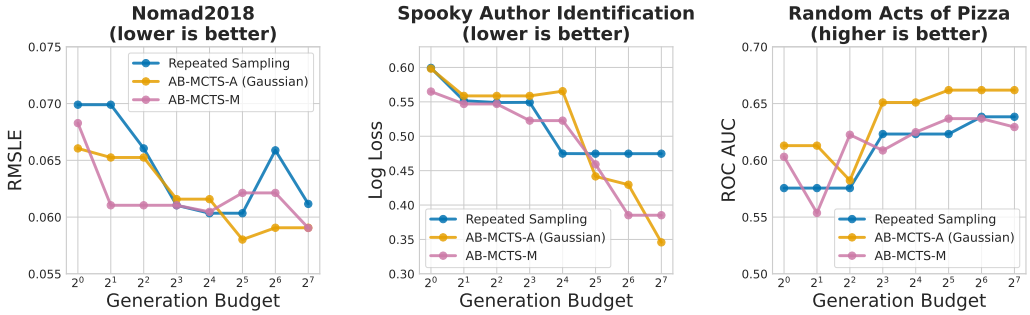

Figure 3: **Performance Comparison on three MLE-Bench tasks.** Each plot shows performance versus the total generation budget. For *Nomad2018 Predicting Transparent Conductors* and *Spooky Author Identification*, lower scores are better (RMSLE and Log Loss, respectively); for *Random Acts of Pizza*, higher is better (ROC AUC). At each budget, we choose the single solution based on validation-set performance and report its test-set score.

action is UCT (Kocsis & Szepesvári, 2006), whereas here we replace it with Thompson Sampling. Let N be a node with potential actions $A = \{a_0, a_1, \ldots, a_{n_{\text{child}}}\}$, where a_0 is the GEN node, and $a_1, \ldots, a_{n_{\text{child}}}$ correspond to already-existing child nodes. Suppose $P(r \mid T, a_i)$ is the posterior predictive distribution over the score r if we choose action a_i. Then Thompson Sampling proceeds by drawing r_{a_i} from $P(r \mid T, a_i)$ for each action a_i and selecting $\hat{a} = \arg\max_{a_i \in A} r_{a_i}$. A key question is how to build the posterior $P(r \mid T, a_i)$, especially for a_0 (the GEN node, which may have no observed data). We propose two strategies: a mixed Bayesian model (AB-MCTS-M) and a node aggregation method (AB-MCTS-A).

AB-MCTS-M leverages a mixed-effects model to separate the variance from newly generated solutions (GEN) and their refinements. Each child group (all children of the same parent) shares a random intercept that captures baseline solution quality, with an additional noise term modeling individual refinements. Observed node scores continuously update this model via MCMC, enabling Thompson Sampling to decide when to expand versus refine. Please refer to Appendix A.4 for more details on AB-MCTS-M.

AB-MCTS-A partitions expansion (GEN) and refinement (CONT) into two distinct posterior distributions. Newly created nodes initialize under the GEN posterior, while subsequent improvements to existing nodes accumulate under CONT. By using Beta or Gaussian conjugate priors, AB-MCTS-A efficiently updates these distributions. Appendix A.5 provides more details of AB-MCTS-A.

3 Experiments

We evaluated our approach on two benchmarks: Code Contest for competitive programming tasks and MLE-Bench for machine learning engineering challenges. Detailed experimental setup, hyperparameters, and comprehensive results can be found in Appendix B.

For Code Contest evaluation, we used GPT-4o and DeepSeek-V3 models, comparing our AB-MCTS variants against two baselines: repeated sampling and standard MCTS. As shown in Figure 2, both AB-MCTS-M and AB-MCTS-A consistently outperform the baselines as the generation budget increases, with AB-MCTS-A (Beta) achieving the highest success rate at maximum budget.

For MLE-Bench experiments, we selected three representative tasks and evaluated them using DeepSeek-V3. As demonstrated in Figure 3, our AB-MCTS variants showed superior performance compared to repeated sampling across most tasks, particularly when given larger generation budgets. AB-MCTS-A (Gaussian) emerged as the strongest performer overall, while AB-MCTS-M demonstrated faster initial convergence in several scenarios.

These results validate the effectiveness of our adaptive branching strategy across different domains and model architectures. The performance gains become particularly pronounced with larger generation budgets, suggesting that AB-MCTS makes more efficient use of additional computational resources compared to traditional approaches.

REFERENCES

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.

Sébastien Bubeck, Rémi Munos, and Gilles Stoltz. Pure exploration in multi-armed bandits problems. In *Algorithmic Learning Theory: 20th International Conference, ALT 2009, Porto, Portugal, October 3-5, 2009. Proceedings 20*, pp. 23–37. Springer, 2009.

Jun Shern Chan, Neil Chowdhury, Oliver Jaffe, James Aung, Dane Sherburn, Evan Mays, Giulio Starace, Kevin Liu, Leon Maksin, Tejal Patwardhan, et al. Mle-bench: Evaluating machine learning agents on machine learning engineering. *arXiv preprint arXiv:2410.07095*, 2024.

DeepSeek-AI. Deepseek-v3 technical report, 2024. URL https://arxiv.org/abs/2412.19437.

Bofei Gao, Feifan Song, Zhe Yang, Zefan Cai, Yibo Miao, Qingxiu Dong, Lei Li, Chenghao Ma, Liang Chen, Runxin Xu, et al. Omni-math: A universal olympiad level mathematic benchmark for large language models. *arXiv preprint arXiv:2410.07985*, 2024.

Ryan Greenblatt. Getting 50% (sota) on arc-agi with gpt-4o. https://www.lesswrong.com/posts/Rdwui3wHxCeKb7feK/getting-50-sota-on-arc-agi-with-gpt-4o, 2024. Accessed: January 21, 2025.

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.

Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *European conference on machine learning*, pp. 282–293. Springer, 2006.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022.

Zhenwen Liang, Ye Liu, Tong Niu, Xiangliang Zhang, Yingbo Zhou, and Semih Yavuz. Improving llm reasoning through scaling inference computation with collaborative verification, 2024. URL https://arxiv.org/abs/2410.05318.

OpenAI. Gpt-4o system card, 2024. URL https://arxiv.org/abs/2410.21276.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models, 2023. URL https://arxiv.org/abs/2203.11171.

An Yang, Beichen Zhang, Binyuan Hui, Bofei Gao, Bowen Yu, Chengpeng Li, Dayiheng Liu, Jianhong Tu, Jingren Zhou, Junyang Lin, Keming Lu, Mingfeng Xue, Runji Lin, Tianyu Liu, Xingzhang Ren, and Zhenru Zhang. Qwen2.5-math technical report: Toward mathematical expert model via self-improvement, 2024. URL https://arxiv.org/abs/2409.12122.

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, 2023.

A METHOD DETAILS

A.1 PROBLEM SETUP

In this section, we define the problem setting and introduce the mathematical notation.

We consider problems where the input is a natural language prompt t_{in} (which may include few-shot examples, task instructions, etc.). Given t_{in}, an LLM produces a natural language output t_{out}, which is then scored by a score evaluator to yield a final score r. We assume $0 \leq r \leq 1$, where higher values of r correspond to better answers.

This two-stage pipeline can be expressed as:

$$r = R(t_{\text{out}}) = R(f_{\text{LLM}}(t_{\text{in}})), \tag{1}$$

where the function f_{LLM} represents an LLM that generates the answer, and R is the score evaluator. f_{LLM} is stochastic and may produce different outputs t_{out} for the same input t_{in}. Here we allow t_{out} to include information beyond the direct answer (e.g. the reasoning steps), and we assume that R properly performs the parsing of this answer to perform the score evaluation.

Our framework applies to any task whose final answer can be quantitatively scored, such as programming tasks (Li et al., 2022; Jain et al., 2024), math problems (Gao et al., 2024), and machine learning competitions (Chan et al., 2024). We assume the score evaluator R is already defined in a task-specific manner, and our goal is to find t_{out} that attains as high an r as possible during the inference-time answer search.

In some tasks that mimic real competitions, the ground-truth score evaluator R_{gt} (reflecting the correctness of the answer) is not accessible during the answer search stage. For example, in MLE-Bench (Chan et al., 2024), the test dataset used for final evaluation is withheld, and in coding competition tasks (Li et al., 2022; Jain et al., 2024), participants can often only submit their code a limited number of times to obtain the score on hidden test cases. In such situations, to search for the best answer, one may resort to a different score evaluator. For instance, in MLE-Bench, this could be the performance on a public dataset, while in coding competitions it might be the fraction of solved public test cases. For mathematical tasks, a separately trained reward model (Yang et al., 2024) may be used. Throughout this work, we assume there is some accessible score evaluator at the answer search stage that can assess the quality of t_{out}.

A.2 EXISTING METHODS FOR INFERENCE-TIME ANSWER SEARCH

We now review two standard approaches that focus on exploration or exploitation alone.

Only Go Wide: Repeated Sampling. A straightforward approach to inference-time answer search is to repeatedly sample an answer from the LLM with a nonzero temperature. We refer to each sampling step as the *direct answer generation process*. By performing it n times,

$$t_{\text{out}}^{m} = f_{\text{LLM}}(t_{\text{in}}), \quad m \in \{1, \ldots, n\}, \tag{2}$$

we obtain multiple candidate answers. Then, one can select the best answer based on a predefined criterion, such as the highest score r (best-of-N), majority voting or self-consistency. Brown et al. (2024) recently showed that as n increases, the coverage of generated answers improves. A similar approach was employed by AlphaCode (Li et al., 2022) to achieve human-level performance on competitive programming tasks.

Only Go Deep: Sequential Refinement. Alternatively, we can leverage the answer refinement process and apply it sequentially to perform the answer search. We consider the situation where we already let some answer generator solve the problem at hand k times, and collected the input-output pairs t_{in}^{j} and t_{out}^{j} where $j \in \{1, \ldots, k\}$ for those answer generations.

We define the *answer refinement process* as a two-step procedure: (1) creation of a new refinement input from the existing input-output pairs, and (2) generation of a new answer $\tilde{t}_{\text{out}}$ from $\tilde{t}_{\text{in}}$. Symbolically,

$$\tilde{t}_{\text{out}} = f_{\text{LLM}}(\tilde{t}_{\text{in}}) = f_{\text{LLM}}\left(h_{\text{refine}}\left(\{t_{\text{in}}^{j}, t_{\text{out}}^{j}\}_{j \in \{1,\ldots,k\}}\right)\right), \tag{3}$$

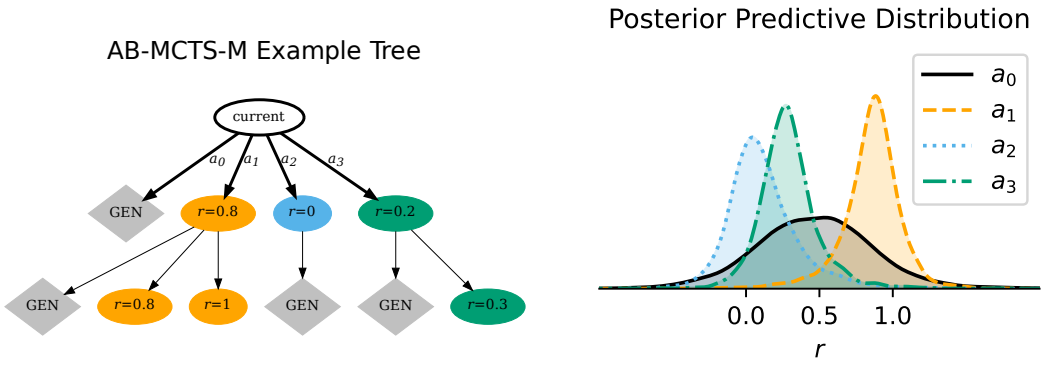

Figure 4: **Example tree structure and the score posterior predictive distribution for AB-MCTS-M.** Here, a_1 leads to a set of child nodes with higher scores, causing a peak at larger r. As more child samples are collected, the variance of the distribution decreases.

where h_{refine} is a refinement input generator that provides all the information necessary for refinement, such as feedback on each answer (e.g. code execution results or errors for coding tasks).

Applying this refinement step iteratively yields n answers:

$$t^1_{\text{in}} = t_{\text{in}}, \tag{4}$$

$$t^1_{\text{out}} = f_{\text{LLM}}(t^1_{\text{in}}), \tag{5}$$

$$t^m_{\text{in}} = h_{\text{refine}}\big(\{t^j_{\text{in}}, t^j_{\text{out}}\}_{j\in\{1,\ldots,m-1\}}\big) \tag{6}$$

$$t^m_{\text{out}} = f_{\text{LLM}}(t^m_{\text{in}}) \tag{7}$$

for $m \in \{2, \ldots, n\}$. Finally, one selects the best candidate among $\{t^a_{\text{out}}\}_{a\in\{1,\ldots,n\}}$ based on a chosen criterion, similar to the Repeated Sampling approach.

A.3 Adaptive Branching MCTS

We can regard the two methods in the previous section (pure exploration and pure exploitation) as special cases of a tree search: the former expands only from the root node, while the latter continues from the most recently reached leaf node on a single linear path, exploring it in depth without branching outward. Here, we generalize both approaches under the framework of Monte Carlo Tree Search (MCTS)[1].

We consider constructing a tree T where each node N corresponds to an input text t_{in} and its LLM-generated output $t_{\text{out}} = f_{\text{LLM}}(t_{\text{in}})$. The expansion from the root does the direct answer generation, while the expansion from a node N employs the answer refinement using the input-output pairs of N's ancestors, including N itself.

We formulate MCTS iteratively as follows. We begin with a single root node and expand the tree n_{nodes} times, ultimately producing $1 + n_{\text{nodes}}$ nodes in total. Each iteration proceeds in three steps:

1. **Selection:** choose a node N in the current tree for expansion.
2. **Expansion:** apply direct answer generation or answer refinement to N to create a new node N_{new}, appended as a child of N.
3. **Score backup:** propagate the score of N_{new} up the tree to update the ancestors' score information, including N_{new} itself.

In our setting, we do not perform a separate rollout, because each node's score r can be evaluated directly once an output is generated. After n_{nodes} expansions, we select the best node based on a chosen criterion.

[1]Unlike conventional MCTS settings, our objective is to maximize the best (rather than cumulative) score, which might suggest focusing solely on exploration. However, with finite action budgets, some exploitation is still needed to achieve the best result (Bubeck et al., 2009), so MCTS remains effective.

Algorithm 1 Adaptive Branching MCTS

function AB-MCTS(n_{nodes})
  $T \leftarrow$ INITIALIZETREE( )
  for $n = 1, \ldots, n_{\text{nodes}}$ **do**
    $N \leftarrow$ SELECTEXPANSIONTARGET(T) ▷ ***Step 1.*** *Selection of an expansion target*
    $N_{\text{new}} \leftarrow$ EXPAND(N, T) ▷ ***Step 2.*** *Expand the selected node to generate a child*
    SCOREBACKUP(N_{new}, T) ▷ ***Step 3.*** *Backup the score from the generated node*
  return SELECTBEST(T)

function SELECTEXPANSIONTARGET(T)
  $N \leftarrow$ GETROOT(T)
  while not ISLEAF(N) **do**
    $N_{\text{next}} \leftarrow$ SELECTCHILD(N, T) ▷ *Detailed in Sec A.4 and A.5*
    if ISGENNODE(N_{next}) **then** ▷ *If GEN node is selected, branch off from the node*
      break
    $N \leftarrow N_{\text{next}}$
  return N

The backup step ensures that the score of N_{new} is shared with its ancestors, enabling more informed node selection in subsequent iterations. We adopt different backup rules in our proposed methods (see Sec A.4 and A.5).

Difficulty in Balancing the Breadth and Depth: Unbounded Branching. Unlike the standard tasks typically tackled by MCTS where the number of possible actions at each node is finite, each call to f_{LLM} can yield a new output even for the same input, making each node's branching factor theoretically infinite. Recent studies (Brown et al., 2024) suggest that drawing more samples from the same prompt often boosts performance, so limiting the branching factor might degrade overall results. To highlight this generative capacity, we introduce a "*GEN node*" that explicitly represents the action of generating a new child. GEN node exists as a child of all the nodes including the newly expanded ones, and represents the action of generating a new child and append it to the GEN node's parent node. Algorithm 1 outlines MCTS with such GEN nodes, which we call *Adaptive Branching Monte Carlo Tree Search* (*AB-MCTS*).

To employ MCTS with potentially unbounded branching, we must define a selection policy for when to create new children (i.e., when to choose the GEN node). Because branching toward new children expands the tree in the "breadth" direction, this decision critically affects how the tree grows.

In this work, we propose two methods to address unbounded branching: *AB-MCTS-M* (AB-MCTS with a Mixed model) and *AB-MCTS-A* (AB-MCTS with node Aggregation). Both follow the overall procedure in Algorithm 1 but differ in how they implement node selection via Thompson Sampling.

Modeling the GEN Node's Score Probability. For node selection, we adopt a Bayesian view of each action's score distribution and apply Thompson Sampling. In standard MCTS, the typical metric for choosing an action is UCT (Kocsis & Szepesvári, 2006), whereas here we replace it with Thompson Sampling. Let N be a node with potential actions

$$A = \{a_0, a_1, \ldots, a_{n_{\text{child}}}\}, \tag{8}$$

where a_0 is the GEN node, and $a_1, \ldots, a_{n_{\text{child}}}$ correspond to already-existing child nodes. Suppose $P(r \mid T, a_i)$ is the posterior predictive distribution over the score r if we choose action a_i. Then Thompson Sampling proceeds by:

1. Drawing r_{a_i} from $P(r \mid T, a_i)$ for each action a_i.
2. Selecting $\hat{a} = \arg\max_{a_i \in A} r_{a_i}$.

A key question is how to build the posterior $P(r \mid T, a_i)$, especially for a_0 (the GEN node, which may have no observed data). We propose two strategies: a mixed Bayesian model (AB-MCTS-M) and a node aggregation method (AB-MCTS-A).

A.4 AB-MCTS-M: ADAPTIVE BRANCHING MCTS WITH MIXED MODEL

Motivation. A natural way to model the GEN node's score is to separate the variance stemming from (i) answer generation with GEN node and (ii) subsequent refinements. A mixed-effects model with a random intercept can capture this structure by assigning a random effect to each child's "group."

Algorithm Outline. In AB-MCTS-M, we fit a separate mixed model at each node N of the MCTS tree, that is, at each sub-step within the MCTS selection step. Specifically, let N_j ($j = 1, \ldots, n_{\text{child}}$) denote the direct child nodes of N, and define $T_{\text{sub}}(N_j)$ as the subtree under N_j, including N_j itself. For a newly generated node $\tilde{N}$ where (i) $j = 0$ (i.e. for GEN node) and $\tilde{N}$ is a direct child of N, or (ii) $j = 1, \ldots, n_{\text{child}}$ and $\tilde{N} \in T_{\text{sub}}(N_j)$, we assume:

$$r_{\tilde{N}} = \alpha_j + \sigma_y \epsilon_{\tilde{N}}, \quad \alpha_j = \mu_\alpha + \sigma_\alpha \epsilon_j, \tag{9}$$

$$\epsilon_{\tilde{N}} \sim \mathcal{N}(0,1), \quad \epsilon_j \sim \mathcal{N}(0,1), \tag{10}$$

Here, α_j is a "group-level" intercept capturing the quality of the base solution at N_j, while $\sigma_y \epsilon_{\tilde{N}}$ represents per-instance noise. The GEN node (action a_0) is treated as a newly introduced group without its own direct observations. However, its group-level intercept α_0 is inferred not from the prior alone but rather from the posterior distribution over μ_α and σ_α, which is informed by the other observed data.

In experiments, we set priors on $(\mu_\alpha, \sigma_\alpha, \sigma_y)$ and estimate posteriors using MCMC. We then draw from the posterior predictive distribution for Thompson Sampling (Figure 4). Typically, the GEN node's distribution has higher uncertainty, promoting additional exploration.

Score Backup. Once a node $\tilde{N}$ is created, its observed score is added to the history of $\tilde{N}$ and all its ancestors. This allows AB-MCTS-M to maintain a record of all subtree scores needed for updating the mixed-model posteriors.

A.5 AB-MCTS-A: ADAPTIVE BRANCHING MCTS WITH NODE AGGREGATION

Motivation. Another way to split the observations into (i) current refinement and (ii) subsequent refinement is to introduce an explicit "CONT" node for continuing refinement, separate from the "GEN" node for a brand-new child. This design makes it easier to model these two phases separately.

Algorithm Outline. AB-MCTS-A aggregates all child nodes under a single "CONT" node to represent ongoing refinement. The probability of exploring new children (GEN) versus refining existing ones (CONT) is handled via Thompson Sampling. Specifically, we assign probabilities (or equivalently parameters of posterior distribution) for all the nodes including GEN and CONT nodes.

We propose two families of conjugate priors:

- AB-MCTS-A (Gaussian): uses a normal-inverse-χ^2 prior, suitable if r may lie outside $[0, 1]$.
- AB-MCTS-A (Beta): uses a Beta prior, specialized for scores in $[0, 1]$. Because of its simple parameterization, the Beta approach can be more sample efficient in settings where r is inherently bounded.

Both approaches permit closed-form updates, making them computationally lightweight.

Figure 5 shows a schematic of AB-MCTS-A. When a new node is generated, its observed score is initially "backed up" to the GEN node that created it. For subsequent updates, the score is backed up to the CONT node, allowing us to separate the effect of initial generation from later refinements.

Using these posterior distributions, we perform Thompson Sampling at each step between GEN and CONT nodes. If the GEN node is selected, a new child node is generated; otherwise, one of the existing children is chosen through Thompson Sampling, using the backed-up scores for each child.

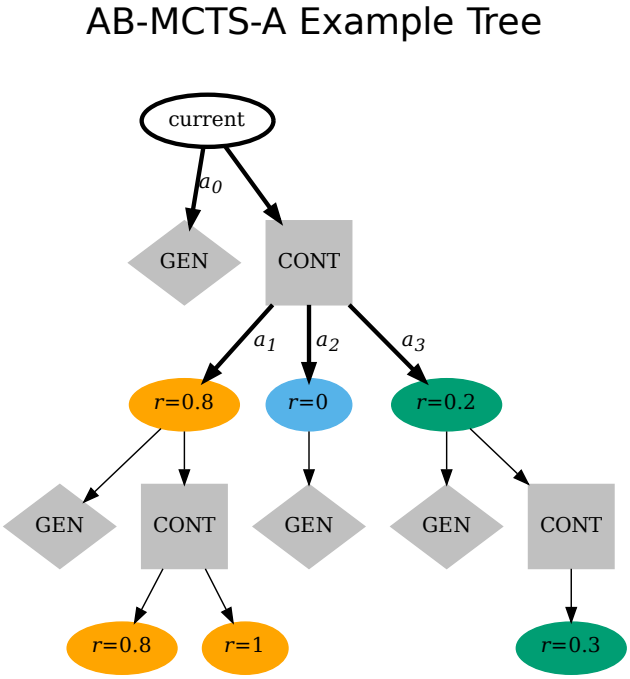

Figure 5: **Example tree structure and the score posterior distribution for AB-MCTS-A.** All child nodes are aggregated under a single "CONT" node.

Finally, because GEN and CONT nodes share the same tree depth in this design, AB-MCTS-A can produce a wider branching structure than AB-MCTS-M. Specifically, generating a node at depth d requires choosing the CONT node (instead of the GEN node) d times in succession, making the probability of reaching greater depths decay approximately geometrically with d.

Score Backup. Whenever a new node is generated, its score is backed up to the GEN node that created it (and all ancestors). In contrast, further refinements of an existing node update the CONT branch's parameters. Figure 5 shows a schematic depiction. Because deeper refinement requires repeatedly selecting CONT, the probability of reaching depth d decays roughly geometrically in practice. Nonetheless, AB-MCTS-A can still explore multiple branches by occasionally choosing the GEN node.

AB-MCTS-A (Gaussian) Parameter Update Rules. Now we describe the parameter update rule in detail. For the Gaussian case, we use a normal-inverse-χ^2 prior:

$$p(r \mid \{r_n\}_{n=1}^N) = \mathcal{N}(\mu \mid \hat{m}, \tfrac{\sigma^2}{\hat{\kappa}})\chi^{-2}(\sigma^2 \mid \hat{\nu}, \hat{\tau}^2), \tag{11}$$

$$\hat{m} = \frac{\breve{\kappa}\breve{m} + N\bar{r}}{\hat{\kappa}}, \tag{12}$$

$$\hat{\kappa} = \breve{\kappa} + N, \tag{13}$$

$$\hat{\nu} = \breve{\nu} + N, \tag{14}$$

$$\hat{\nu}\,\hat{\tau}^2 = \breve{\nu}\,\breve{\tau}^2 + \sum_{n=1}^{N}(r_n - \bar{r})^2 \;+\; \frac{N\,\breve{\kappa}}{\breve{\kappa} + N}\,(\hat{m} - \bar{r})^2, \tag{15}$$

$$\bar{r} = \frac{1}{N}\sum_{n=1}^{N} r_n, \tag{16}$$

where r_n is the observed score.

AB-MCTS-A (Beta) Parameter Update Rules. Alternatively, if $r \in [0, 1]$, we can use a Beta distribution with the following parameter update rules after observing $\{r_n\}_{n=1}^N$:

$$p(r \mid \{r_n\}_{n=1}^N) = B(r \mid \hat{\alpha}, \hat{\beta}), \tag{17}$$

$$\hat{\alpha} = \breve{\alpha} + \sum_{n=1}^{N} r_n, \tag{18}$$

$$\hat{\beta} = \breve{\beta} + \sum_{n=1}^{N} (1 - r_n), \tag{19}$$

where $B(\cdot \mid \alpha, \beta)$ denotes the Beta distribution. We note that, usually this update rule is used in conjunction with the Bernoulli trial, but here we directly use Beta distribution to model the score distribution. In practice, this parameter update rule worked well according to our experimental results.

B Additional Experimental Details and Analysis

B.1 Experimental Setup

Tasks and Datasets. We evaluated our approach on two benchmarks for code generation (Code Contest (Li et al., 2022)) and machine learning engineering (MLE-Bench (Chan et al., 2024)).

Code Contest is a well-established competitive programming benchmark, providing public tests and hidden tests. We use the public tests to calculate each node's score and the hidden tests for final evaluation. A solution is counted as correct only if it passes all hidden test cases for a given problem, and the success rate is defined as the fraction of problems for which the chosen solution is fully correct.

For MLE-Bench, which comprises practical machine learning tasks derived from Kaggle competitions, we adopt three low-complexity challenges to enable fair comparisons while keeping the API cost manageable (Chan et al., 2024). Each competition's training data is randomly split into 80% for training and 20% for validation. The validation set is used to obtain the scores for each node. We select a node with the highest validation score at a given inference budget and then evaluate it on the hidden test set to get the final result. Following previous research (Chan et al., 2024), we use the AIDE scaffold for our experiments.

Models. We perform our experiments using GPT-4o (`gpt-4o-2024-08-06`) (OpenAI, 2024) and DeepSeek-V3 (`deepseek-chat`) (DeepSeek-AI, 2024). Each LLM generates a complete solution in a single pass, and we set a maximum generation budget for each model and task. For Code Contest, we set a generation budget of 512 for DeepSeek-V3 and 128 for GPT-4o. Due to the expenses associated with API usage, we conducted MLE-Bench experiments with DeepSeek-V3, fixing its generation budget at 128. We set the temperature of GPT-4o to 0.6 for all tasks, while for DeepSeek-V3 we use a temperature of 1.5 for Code Contest and 1.0 for MLE-Bench.

Baselines. We compare our proposed AB-MCTS approaches with the following methods:

- **Repeated Sampling (Best-of-N)**: We independently sample up to N candidate solutions from the LLM using a single prompt. This strategy has shown strong performance on coding tasks (Li et al., 2022; Brown et al., 2024).
- **Standard MCTS**: As a search-based baseline, we adopt a MCTS approach with UCT-based node selection. Following the previous work (Zhou et al., 2023), we generate 5 child nodes in each expansion step, producing a total node budget of 128. This provides a non-adaptive tree search baseline for comparison.

AB-MCTS Parameters. We assign the following priors for AB-MCTS-M in Equations 9 and 10:

$$\mu_\alpha \sim \mathcal{N}(0.5, 0.2^2), \quad \sigma_\alpha \sim \mathcal{N}_{\text{half}}(0.2^2), \quad \sigma_y \sim \mathcal{N}_{\text{half}}(0.3^2), \tag{20}$$

where $\mathcal{N}_{\text{half}}$ is the half-normal distribution. The prior parameters are designed such that the prior distribution lies within the range of $[0, 1]$. For AB-MCTS-A (Gaussian), we set $\breve{m} = 0$, $\breve{\kappa} = 1$,

$\breve{\nu} = 1$, and $\breve{\tau}^2 = 0.1$ in Equations 11 - 16, and for AB-MCTS-A (Beta), we set $\breve{\alpha} = 0.5$ and $\breve{\beta} = 0.5$ in Equations 17 - 19. We chose these parameters in a way that imposes as few assumptions as possible, thereby minimizing any potential bias.

B.2 RESULTS

Results on Code Contest. We first evaluate our frameworks on Code Contest. Figure 2 reports the success rate (Pass@1) versus the generation budget for both GPT-4o (left) and DeepSeek-V3 (right). Figure 2 (left) shows that, with GPT-4o, all variants of AB-MCTS exceed Repeated Sampling and Standard MCTS across most budgets, ultimately achieving higher accuracy at the maximum budget of 2^7. In particular, AB-MCTS-M consistently outperforms both baselines at every examined budget, though its final accuracy is slightly lower than that of AB-MCTS-A (Beta). As shown in Figure 2 (right), when using DeepSeek-V3, all algorithms exhibit similar performance for smaller budgets. However, once the budget exceeds 2^6, AB-MCTS-A (Beta) and AB-MCTS-M show notably stronger improvements than Repeated Sampling or Standard MCTS. AB-MCTS-A (Gaussian) increases at a slightly slower rate, but it ultimately outperforms the baselines. Overall, these results suggest that, for both GPT-4o and DeepSeek-V3, scaling up the generation budget yields greater performance gains for AB-MCTS compared to the baselines, confirming the effectiveness of our adaptive branching search algorithms.

Results on MLE-Bench. In Figure 3, we compare AB-MCTS-A (Gaussian) and AB-MCTS-M with Repeated Sampling on three competitions from MLE-Bench. In *Nomad2018* (Figure 3, left), both variants of AB-MCTS steadily reduce RMSLE relative to Repeated Sampling. Although brief fluctuations appear for moderate budgets (2^4–2^6), AB-MCTS-M and AB-MCTS-A (Gaussian) eventually converge below 0.060 at larger budgets, outperforming the baseline. Using *Spooky Author Identification* (Figure 3, middle), AB-MCTS also exceeds the baseline as the generation budget increases. Between 2^5 and 2^7, AB-MCTS-M and AB-MCTS-A (Gaussian) continue to improve their scores, while Repeated Sampling plateaus near 0.47. Lastly, for *Random Acts of Pizza* (Figure 3, right), AB-MCTS-A (Gaussian) begins to exceed the other methods from around 2^3 and ultimately surpasses 0.66 at higher budgets. Meanwhile, AB-MCTS-M eventually falls slightly behind Repeated Sampling, yet remains competitively close overall.

Taken together, these MLE-Bench results mirror our coding-task results: both AB-MCTS variants generally outperform Repeated Sampling once the budget is sufficiently large. Although AB-MCTS-A (Gaussian) ultimately turns out to be the strongest framework in these three tasks, AB-MCTS-M often scores well in earlier stages, suggesting that it can converge more rapidly in some scenarios. Collectively, these results underline the advantages of adaptively deciding when to explore new solutions versus refining promising ones.