Can I Have Your Order? Monte-Carlo Tree Search for Slot Filling Ordering in Diffusion Language Models

Anonymous Authors[1]

Abstract

While plan-and-infill decoding in Masked Diffusion Models (MDMs) shows promise for mathematical and code reasoning, performance remains highly sensitive to slot infilling order, often yielding substantial output variance. We introduce DIFFUSEARCH, a framework that formulates slot selection as decision making and optimises infilling orders through Monte Carlo Tree Search (MCTS). DIFFUSEARCH uses look-ahead simulations to evaluate partial completions before commitment, systematically exploring the combinatorial space of generation orders. Experiments show an average improvement of 3.2% over autoregressive baselines and 8.0% over baseline plan-and-infill, with notable gains of 19.5% on MBPP and 4.9% on MATH500. Our analysis reveals that while DIFFUSEARCH predominantly follows sequential ordering, incorporating non-sequential generation is essential for maximising performance. We observe that larger exploration constants, rather than increased simulations, are necessary to overcome model confidence biases and discover effective orderings. These findings establish MCTS-based planning as an effective approach for enhancing generation quality in MDMs.

1. Introduction

While autoregressive models (ARMs; Liu et al. 2024; Yang et al. 2025a) have achieved remarkable progress across a wide range of reasoning tasks (Wei et al., 2022; Leang et al., 2025a; Lyu et al., 2023), their inference is fundamentally limited by the sequential constraint of autoregressive decoding. Masked Diffusion Models (MDMs; Bie et al. 2025; Ye et al. 2025) have been proposed to overcome the limitations of current ARMs through an iterative denoising process with no fixed generation order. MDMs enable non-sequential order decoding by assuming conditional independence among target tokens and offer the potential to discover generation orders beyond rigid left-to-right trajectories (Li et al., 2023; Kim et al., 2025; Fu et al., 2025). While MDMs offer inference efficiency, they often underperform compared to ARMs (Nie et al., 2025). One critical reason is that such generation can lead to interdependent tokens generated simultaneously without mutual conditioning, making output quality sensitive to the generation order (Li et al., 2025).

Recently, the "plan-and-infill" framework (Li et al., 2025) has emerged as a promising approach for optimising generation orders in MDMs, where each denoising iteration comprises two steps: *a planning phase* that selects a sub-sequence (slot) from the set of currently masked subsequences, and *an infilling phase* that generates tokens within this selected slot autoregressively. While this framework alleviates the difficulty of generating tokens with strong local, inner-slot dependency (Li et al., 2025), the inter-slot generation order remains critical to output quality, as errors in ordering can induce unmodelled dependencies that propagate across iterations and undermine global coherence. Identifying such orderings is challenging due to the vast combinatorial space of possible slot permutations. Consequently, heuristic planners, such as confidence-based selection used in current MDMs (Ye et al., 2025; Schuster et al., 2022), often fail to account for long-range dependencies, leading to performance degradation.

To overcome the difficulty of reliably selecting slots for infilling (i.e., *the planning phase*), we introduce DIFFUSEARCH, a training-free framework that formulates slot selection as a decision-making problem. Rather than seeking a provably optimal schedule (Figure 1), DIFFUSEARCH performs targeted lookahead over the combinatorial space of slot orderings using Monte Carlo Tree Search (MCTS; Kocsis & Szepesvári 2006) with prior-guided expansion (Silver et al., 2017). In particular, the search integrates the model's intrinsic confidence scores as prior probabilities and uses a hybrid reward mechanism combining immediate denoising quality with rollout-based estimation of long-term trajectory coherence. By simulating future generation steps, DIFFUSEARCH identifies slot orderings that reduce error propagation

[1]Anonymous Institution, Anonymous City, Anonymous Region, Anonymous Country. Correspondence to: Anonymous Author <anon.email@domain.com>.

Preliminary work. Under review by the International Conference on Machine Learning (ICML). Do not distribute.

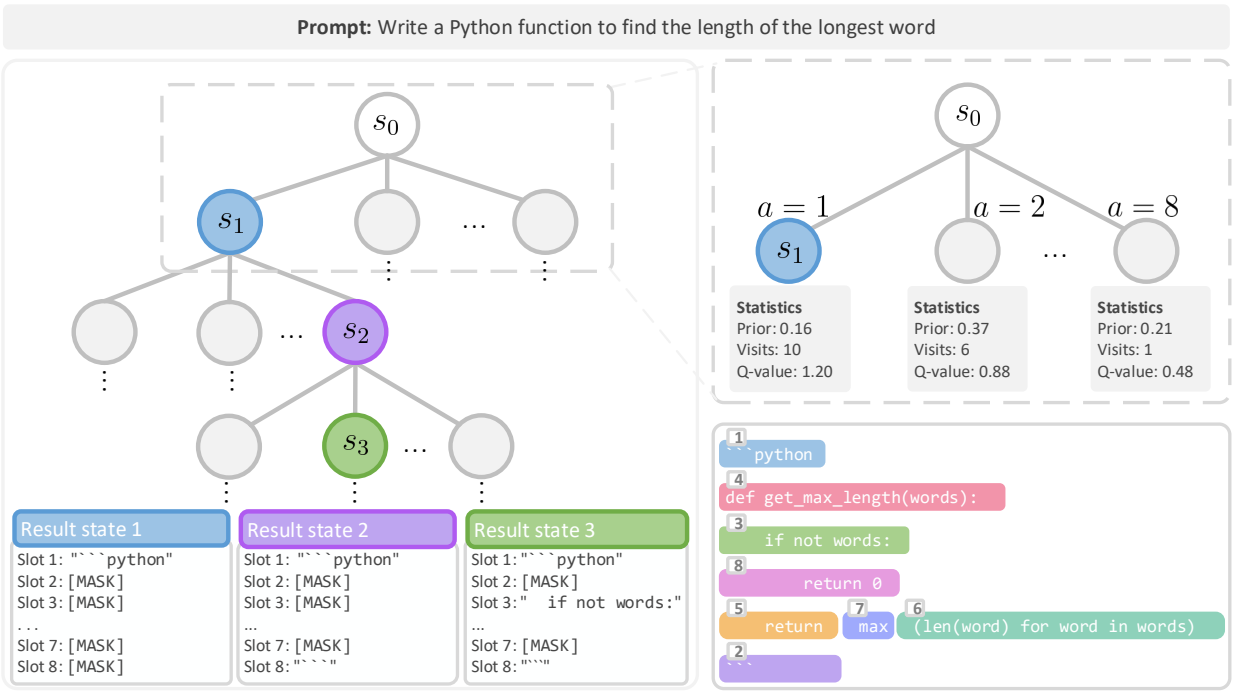


Figure 1. Overview of DIFFUSEARCH. We formulate slot selection as a sequential decision-making process optimised via Monte Carlo Tree Search. As illustrated in the *Statistics* box, the model's greedy prior ($P = 0.37$) favours immediately generating the function definition (i.e., slot 2: "`def get_max_length(words):`"). However, through look-ahead simulations, the search algorithm discovers that starting with the syntax declaration (i.e., slot 1: "```` ```python ````") yields a higher long-term value (i.e., $Q = 1.20$ for slot 1 vs. $Q = 0.88$ for slot 2), allowing the model to override the biased local prior and ensure global coherence.

across iterations, thereby improving generation quality.

We evaluate DIFFUSEARCH across six reasoning benchmarks, where it consistently outperforms current MDMs and the state-of-the-art "plan-and-infill" method, ReFusion (Li et al., 2025). Specifically, DIFFUSEARCH achieves absolute accuracy gains of 19.45% on MBPP (Austin et al., 2021) and 4.9% on MATH500 (Hendrycks et al., 2021). DIFFUSEARCH narrows the performance gap between diffusion and autoregressive models, matching or exceeding autoregressive performance on five out of six reasoning benchmarks under identical experimental conditions. These results demonstrate the critical importance of strategic slot planning in MDMs and confirm the effectiveness of DIFFUSEARCH's trajectory exploration approach.

We further provide a comprehensive analysis revealing two key insights into DIFFUSEARCH's success. (1) While DIFFUSEARCH predominantly adheres to sequential (left-to-right) generation, it strategically deviates to non-sequential orderings for a critical subset of samples, achieving higher accuracy on these cases and demonstrating that *incorporating non-sequential generation is essential for maximising performance on reasoning tasks*. (2) Unlike traditional MCTS (Guan et al., 2025; Yang et al., 2025b) applied to ARMs, our approach prioritises exploration over simulation depth. We find that an increasing number of simulations does not consistently improve performance. Instead, a large exploration constant is necessary to overcome the model's confidence priors and discover effective slot orderings, indicating that the primary challenge in slot planning lies in initiating search down low-prior branches to ensure sufficient exploration breadth, escaping bias towards locally confident but globally incoherent initial priors (Section 6).

In summary, our contributions are: 1) we propose DIFFUSEARCH, a training-free framework that tailors MCTS specifically for searching slot orderings in MDMs, effectively closing the performance gap with ARMs; 2) we provide empirical analysis revealing that strategic non-sequential generation is critical to performance, validating MCTS as an effective approach for slot orderings; 3) we show that exploration breadth is critical to slot planning, requiring a large exploration constant to escape biased confidence priors and discover better orderings.

2. Background

Markov Decision Process. Sequential decision-making problems are typically modelled as a *Markov Decision Process* (MDP; Bellman 1957). Formally, an MDP is a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R} \rangle$, where $\mathcal{S}$ is the *state space* representing all of the possible configurations of the environment, and $\mathcal{A}$

is the *action space*. At every time step t, an agent in state $s_t \in \mathcal{S}$ selects an action $a \in \mathcal{A}$, upon which the environment transitions to a successor state $s_{t+1} \in \mathcal{S}$ according to the *transition probability function* $\mathcal{T}(s_{t+1} \mid s_t, a)$. The agent receives a reward specified by the *reward function* $\mathcal{R}$, and the objective is to learn a policy that maximises the expected cumulative reward over time.

Monte Carlo Tree Search. *Monte Carlo Tree Search* (MCTS) is a heuristic search algorithm that approximates optimal policies in MDPs by combining tree search with Monte Carlo simulation to balance exploration and exploitation (Coulom, 2006). In the tree, each node represents a state $s \in \mathcal{S}$ and each edge represents an action $a \in \mathcal{A}$. The algorithm operates through four iterative phases: (i) selection, (ii) expansion, (iii) simulation, and (iv) backpropagation.

During (i) *selection*, the search starts from the root node corresponding the current state and traverses the tree by selecting child nodes according to a bandit-based policy. Upon reaching a leaf node, the (ii) *expansion* step instantiates child nodes corresponding to previously unexplored actions available at that state. One of these newly created children is then selected for evaluation. To estimate its value, a (iii) *simulation* (or *rollout*) is performed by sampling a trajectory from the current state until a terminal condition is met, often using a user-defined lightweight policy to guide action selection. Finally, during (iv) *backpropagation*, the cumulative reward resulting from the simulation is propagated backward up the tree. A common policy choice used during the selection phase is the Upper Confidence Bounds for Trees (UCT; Kocsis & Szepesvári 2006):

$$\mathrm{UCT}(s,a) = Q(s,a) + c_{\text{uct}}\sqrt{\frac{\ln N_s}{N_s^a}}, \tag{1}$$

where $Q(s,a)$ is the empirical mean return over all rollouts in which action a was selected from state s, N_s is the number of visits of state s, and N_s^a is the number of times action a was selected from s. Visit counts are initialised to zero upon node instantiation and are updated at each iteration of the algorithm during backpropagation. The constant $c_{\text{uct}} > 0$ controls the exploration–exploitation trade-off, i.e., a larger c_{uct} value corresponds to higher exploration.

Neural-guided MCTS. In high-dimensional action spaces, modern MCTS variants such as AlphaZero (Silver et al., 2017) replace stochastic rollouts with learned value functions and incorporate neural policy priors to guide exploration. These methods typically employ the *Predictor-Upper Confidence Tree* (PUCT) algorithm which selects children using the criterion defined as:

$$\mathrm{PUCT}(s,a) = \underbrace{Q(s,a)}_{\text{Exploitation term}} + \underbrace{c \cdot P(s,a) \cdot \frac{\sqrt{N_s}}{1+N_s^a}}_{\text{Exploration term}}, \tag{2}$$

where $P(s,a)$ denotes the prior probability of selecting action a in state s, as predicted by a neural policy π_θ. The policy prior might be learnt independently (e.g., via reinforcement or imitation learning (Silver et al., 2017; Granter et al., 2017) or jointly with MCTS by training the network to match the search-induced action distribution (Schrittwieser et al., 2020). The exploration term biases early search toward actions with high prior probability while ensuring that the influence of the prior diminishes as visit counts increase. This mechanism effectively reduces the search effort by focusing exploration on semantically plausible actions and is particularly well suited for inference-time optimisation. The definition of $Q(s,a)$, N_s, N_s^a and c is as in Equation 1.

3. DIFFUSEARCH

In this section, we formulate the slot planning as an MDP (Section 3.1), and introduce our MCTS-based slot planning approach for determining infilling orders (Section 3.2).

3.1. Problem Formulation

Suppose we have a generative model with parameters θ that produces text as a sequence $\mathbf{x} = (\mathbf{x}^1, \ldots, \mathbf{x}^K)$ of K contiguous and disjoint *slots*, i.e., strings each with L tokens. While tokens within each slot are generated autoregressively, slots themselves may be generated in an arbitrary order.

Our goal is to determine an ordering of the slots that reflects how the generation of one slot influences the others. We represent such ordering as a permutation $(\sigma_1, \sigma_2, \ldots, \sigma_K)$, where σ_t is the index of the slot generated at step t, with $t = 1, \ldots, K$. We formulate this optimisation problem as a deterministic MDP $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R} \rangle$, whose components are defined below.

State space ($\mathcal{S}$): A state encodes which slots have been generated so far and the corresponding partially filled sequence. Formally, a state $s_t \in \mathcal{S}$ visited at step t is defined as a pair $(\sigma_{0:t}, \mathbf{x}_t)$, where $\sigma_{0:t} = (\sigma_1, \ldots, \sigma_t)$ represents the prefix of σ containing the indexes of the slots generated up to step t, while $\mathbf{x}_t = (\mathbf{x}_t^1, \ldots, \mathbf{x}_t^K)$ is the current (possibly partially filled in) text sequence. For any slot index $k \notin \{\sigma_1, \ldots, \sigma_t\}$, the corresponding slot $\mathbf{x}_t^k$ is populated with the special `[MASK]` token. The first state ($t = 0$) is initialised with $\sigma_{0:0} = \emptyset$[1] and $\mathbf{x}_0 =$ `[MASK]`^{K}, where `[MASK]`^{K} indicates a sequence of K `[MASK]` tokens.

Action space ($\mathcal{A}$): An action corresponds to selecting the index of one of the masked slots. Hence, for every $a \in \mathcal{A}$, $a \in \{1, \ldots, K\}$. At step t, the admissible action set will be $\mathcal{A}_t = \{k \mid k \in \{1, \ldots, K\} \setminus \{\sigma_1, \ldots, \sigma_t\}\}$.

Example 3.1. Consider Figure 1, where the following prompt is given *"Write a Python function to find the length*

[1]We use $\emptyset$ to denote an empty sequence.

of the longest word", and where the target response is partitioned into $K = 8$ slots (e.g., slot 1: "```` ```python`", slot 2: "`def get_max_length(words):`", etc.). The initial state is s_0, where all slots are masked. If slot 1 is selected, then the action is $a = 1$ and the successor state becomes s_1, where slot 1 is infilled (e.g., with "```` ```python`") and the other slots are still masked, as we can also see on the bottom left side of the Figure in the result of state s_1. ◁

Transition function ($\mathcal{T}$): In our formulation the transition function is deterministic, and hence, given a state s_t and an action $a = \sigma_{t+1}$, $\mathcal{T}(s_{t+1} \mid s_t, a)$ assigns probability 1 to a unique successor state. Since the slot chosen by the action a is infilled via argmax decoding of the generative model, $s_{t+1} = (\sigma_{0:t} \oplus \sigma_{t+1}, \mathbf{x}_{t+1})$, where $\oplus$ is used to indicate that we concatenate the index σ_{t+1} to the sequence $\sigma_{0:t}$, while the updated sequence $\mathbf{x}_{t+1}$ is identical to $\mathbf{x}_t$ except at position σ_{t+1}, in which the mask is replaced by the model's predicted tokens.

Reward function ($\mathcal{R}$): In the absence of ground-truth references during inference, we rely on the model's intrinsic confidence estimates to guide the tree search. Following recent work on confidence-based generation showing confidence correlates positively with downstream accuracy (Leang et al., 2025b; Nie et al., 2025; Prabhudesai et al., 2025), we define rewards based on the probability the model assigns to its own predictions. We define the reward as the *slot-level confidence*:

$$\mathcal{R}(s_t, a) = \frac{1}{L} \sum_{i=1}^{L} \mathbb{P}_\theta(\mathbf{x}_{t+1}^{\sigma_{t+1}}[i] \mid \mathbf{x}_t, \text{Input Prompt}), \quad (3)$$

where $\mathbf{x}_{t+1}^{\sigma_{t+1}}[i]$ denotes the i-th token of the slot at index σ_{t+1}, and L is the slot size. At every step t, the reward $\mathcal{R}(s_t, a)$ thus quantifies the generation quality slot filled following the action a (i.e., the one at index σ_{t+1}) as the mean probability assigned by the generative model across all tokens within that slot:

3.2. Slot Planning via Monte Carlo Tree Search

To solve the MDP defined above without training auxiliary networks, we adapt MCTS to guide slot ordering by designing a rollout-based confidence estimator that captures future coherence beyond immediate slot-level confidence rather than relying on greedy local heuristics. We describe the four phases of the resulting MCTS procedure below.

Selection. Starting from the root node s_0, the algorithm recursively traverses the search tree by selecting actions according to the PUCT criterion in Equation (2) until an unexpanded leaf node is reached. For previously unvisited state–action pairs, the corresponding value estimates are initialised as $Q(s, a) = 0$.

Example 3.2. *(cont'd Example 3.1)* At the root s_0, the algorithm compares the values of starting with different slots. Figure 1 illustrates a clear divergence between the model's priors and the search values. Indeed, the prior $P(s, a)$ initially favours starting with the function definition (slot 2, with prior 0.37), likely due to it containing the core semantics of the prompt. However, the Q-values accumulated from the look-ahead simulations favour starting with the syntax declaration (slot 1). Despite having a lower prior than slot 2, slot 1 yields a much higher Q-value (of 1.20) compared to slot 2 (with 0.88 Q-value). Thus, the PUCT-based policy outweighs the greedy prior and selects action $a = 1$ (which infills slot 1 with "```` ```python`"). This choice captures the dependency that establishing the syntax first leads to a more coherent generating trajectory, as the syntax declaration acts as a conditioner for the remaining slots, enforcing code structure and simplifying the calculation of indentation, on which Python is heavily dependent. ◁

Expansion. When an unexpanded leaf node corresponding to state s_t is reached, we expand the node by creating child nodes for all admissible actions $a \in \mathcal{A}_t$. The prior over actions is obtained by a forward pass of the generative model conditioned on the current state s_t, from which we extract the slot-level confidence score $\mathcal{R}(s_t, a)$ for each unfilled slot. These scores are normalised to form the action prior:

$$P(a \mid s_t) = \frac{\mathcal{R}(s_t, a)}{\sum_{a' \in \mathcal{A}_t} \mathcal{R}(s_t, a')}, \quad (4)$$

Following expansion, a child node is selected according to the PUCT criterion to initiate the simulation phase.

Example 3.3. *(cont'd Example 3.2)* After reaching the leaf node s_1 (at which only slot 1 has been infilled), the algorithm expands the tree by creating child nodes for the remaining unfilled slots $\{2, 3, \ldots, 8\}$. The priors $P(a \mid s_1)$ are recalculated based on the filled syntax declaration. To this end, the model likely assigns higher prior probability to slot 2 (i.e., the function header) or slot 3 (which checks a corner case). ◁

Simulation. To evaluate an expanded leaf corresponding to state s_t, we estimate the value of taking action a by combining the immediate slot-level reward with a stochastic rollout estimate:

$$V(s_t, a) = \lambda \cdot \mathcal{R}(s_t, a) + (1 - \lambda) \cdot \mathbb{E}_{\pi_{\text{roll}}}[G]. \quad (5)$$

The mixing coefficient λ balances immediate local confidence against long-term trajectory quality. This balance is important as evaluating a leaf node solely based on the immediate confidence $\mathcal{R}(s_t, a)$ can be misleading: a slot that appears highly confident in isolation may constrain the model's predictions for remaining slots, leading to poor overall generation quality. The rollout term G approximates the remaining return by continuing slot infilling from the successor state $s_{t+1} = \mathcal{T}(s_t, a)$ until a terminal state is

Algorithm 1 Simulation Step

Input: Current state s_t, temperature τ, mixing coefficient λ, slot-level confidence scores $\{\mathcal{R}(s_t, a)\}_{a \in \mathcal{A}_t}$
Output: Rollout estimate
$s_r \leftarrow s_t$ *// Copy state for lookahead*
$T \leftarrow 0, G \leftarrow 0$
while s_r has unfilled slots **do**
  $\mathcal{A}_r \leftarrow \{$indices of unfilled slots in $s_r\}$
  // Stochastic selection based on temperature
  Compute probabilities $\pi_{\text{roll}}(a \mid s_r) \propto \exp(\mathcal{R}(s_r, a)/\tau)$ for all $a \in \mathcal{A}_r$
  Sample next slot action $\tilde{a} \sim \{\pi_{\text{roll}}(a \mid s_r)\}$
  $G \leftarrow G + \mathcal{R}(s_r, \tilde{a})$
  $s_r \leftarrow \mathcal{T}(s_r, \tilde{a})$ by filling slot $\tilde{a}$ *// Simulate infill*
  $T \leftarrow T + 1$
end while
$G \leftarrow G/T$ **if** $T > 0$ **else** 0 *// Weighted combination*
return G

reached. At each rollout step r, given a state s_r, we compute $\mathcal{R}(s_r, a)$ for all admissible actions $a \in \mathcal{A}_r$ and sample the next slot according to a temperature-scaled (according to τ) softmax policy:

$$\pi_{\text{roll}}(a \mid s_r) = \frac{\exp(\mathcal{R}(s_r, a)/\tau)}{\sum_{a' \in \mathcal{A}_r} \exp(\mathcal{R}(s_r, a')/\tau)}. \quad (6)$$

The trajectory score is the average reward accumulated along the rollout,

$$G = \frac{1}{T} \sum_{i=1}^{T} \mathcal{R}(s_i, \tilde{a}_i), \quad (7)$$

where $(s_i, \tilde{a}_i)$ are the state–action pairs encountered during the rollout and T is the rollout length.

Example 3.4. *(cont'd Example 3.3)* From the newly expanded node, the rollout policy stochastically fills the remaining slots to estimate a trajectory value. For example, it might sample the path: fill slot 3 (" if not words:") $\rightarrow$ slot 8 (closing tag "` ` `") $\rightarrow$ slot 2 (function header "`def get_max_length(words):`"). If this specific slot ordering results in high model confidence across all tokens, then a high value $V(s_1)$ is returned. ◁

We provide the pseudocode for the rollout process in Algorithm 1. Further, in Appendix B, we present a concrete numerical example of how the rollout values are calculated.

Backpropagation. After evaluating a leaf node, the estimated value $V(s, a)$ is recursively propagated back to the root along the traversal path. For each visited (state, action) pair (s, a), we update the visit counts

$$N_s \leftarrow N_s + 1, \qquad N_s^a \leftarrow N_s^a + 1, \quad (8)$$

and accumulate the total value associated with the edge (s, a) as

$$W(s, a) \leftarrow W(s, a) + V(s, a). \quad (9)$$

The empirical mean action value is then given by

$$Q(s, a) = \frac{W(s, a)}{N_s^a}, \quad (10)$$

which serves as the exploitation term in the PUCT selection rule.

After N_{sim} simulations, we select the next action at the root of the current search tree, i.e., the current state s_t, using the robust child criterion

$$a_t^* = \arg\max_{a \in \mathcal{A}_t} N_{s_t}^a, \quad (11)$$

which chooses the action most frequently selected during search. While PUCT is used to guide exploration during simulations, the final action selection relies solely on visit counts to avoid bias from exploration bonuses. We execute a_t^* to transition to s_{t+1} and repeat the search from the new root state until all slots are filled.

Example 3.5. *(cont'd Example 3.4)* The high value obtained from constructing a syntactically valid Python function is propagated up the tree. As the last step of our MCTS-based algorithm, the root node updates its statistics for action $a = 1$ (including the mean value estimates and visit counts), reinforcing the starting with the syntax declaration (in Markdown), rather than generating directly the code, yields the preferred generation path. Finally, after N_{sim} simulations are completed, a single child of the root node is selected corresponding to the highest visit count. In the context of Figure 1, this results in selecting the blue node (corresponding to transitioning from state s_0 to s_1 through action $a = 1$), which now marks the syntax declaration as the next slot to be infilled. ◁

While Algorithm 1 details our core value estimation strategy (the rollout), the full algorithm for slot selection is provided in Appendix A. Further, in Appendix F, we provide a concrete, step-by-step walkthrough of the four MCTS phases of our approach, across multiple simulations.

4. Experimental Setup

Evaluation Benchmarks and Metrics. We evaluate on six benchmarks, ranging from mathematical reasoning, code generation, and general knowledge: GSM8K (Cobbe et al., 2021), MATH500 (Hendrycks et al., 2021), MBPP (Austin et al., 2021), HumanEval (Chen, 2021), ARC Challenge (Clark et al., 2018), and GPQA-Diamond (Rein et al., 2024). For all evaluations, we use the Pass@1 metric. To ensure fair comparison, we re-evaluate all baselines using chain-of-thought prompting (Kojima et al., 2022) (i.e., let's think step by step), allowing models to generate intermediate reasoning steps before producing final answers. All results averaged over three independent runs and reported with standard errors. (Details can be referred to Appendix C.)

Models. We use ReFusion (Li et al., 2025) as our base model, as it is, to the best of our knowledge, the only available slot-and-infill diffusion language model. We compare

Model	GSM8K	MATH500	ARC	GPQA-Diamond	MBPP	HumanEval	Average
Autoregressive Models							
Llama-3.1 8B	83.48% ± 0.64%	35.90% ± 1.56%	83.32% ± 0.06%	31.14% ± 2.37%	25.49% ± 1.38%	59.76% ± 0.86%	53.18%
Qwen2.5 7B	88.02% ± 2.25%	46.50% ± 0.71%	87.80% ± 0.12%	16.94% ± 2.49%	64.90% ± 1.22%	71.78% ± 4.22%	62.66%
Qwen3 8B	**88.21%** ± 2.09%	46.20% ± 0.28%	88.40% ± 0.12%	23.13% ± 8.39%	69.81% ± 1.41%	76.24% ± 2.09%	65.33%
Diffusion Models							
LLaDA 8B	84.50% ± 1.73%	36.70% ± 2.12%	84.09% ± 2.11%	21.39% ± 1.20%	32.16% ± 3.49%	35.40% ± 0.00%	49.04%
Dream 7B	84.31% ± 0.22%	42.90% ± 0.42%	84.95% ± 0.05%	25.42% ± 0.04%	53.23% ± 4.00%	57.24% ± 5.06%	58.01%
ReFusion 7B	85.64% ± 0.59%	42.90% ± 0.71%	87.98% ± 0.49%	30.45% ± 0.09%	54.12% ± 6.64%	62.05% ± 1.10%	60.52%
+ Sequential	77.56% ± 2.03%	38.90% ± 0.71%	87.09% ± 0.88%	33.43% ± 0.71%	50.58% ± 0.00%	62.18% ± 2.21%	58.29%
+ Random	66.87% ± 1.93%	29.10% ± 0.71%	80.98% ± 0.60%	27.63% ± 3.54%	21.40% ± 1.65%	32.60% ± 1.13%	43.10%
DIFFUSEARCH	87.91% ± 1.02%	**47.80%** ± 0.85%	**88.68%** ± 0.39%	**34.78%** ± 0.27%	**73.57%** ± 1.61%	**78.37%** ± 0.45%	**68.52%**

Table 1. Performance comparison on reasoning and code generation benchmarks (mean and standard deviation). DIFFUSEARCH achieves the highest average performance, outperforming both autoregressive and diffusion models. Notably, DIFFUSEARCH demonstrates superior performance on MATH500, ARC, GPQA-Diamond, MBPP, and HumanEval, while remaining highly competitive on GSM8K.

DIFFUSEARCH against eight baselines: the vanilla ReFusion model, ReFusion with random slot ordering and sequential slot ordering, LLaDA-8B-Instruct (Nie et al., 2025) and Dream 7B (Ye et al., 2025) as diffusion-based baselines, with Qwen2.5 7B (Team et al., 2024) and Qwen3 8B (Yang et al., 2025a) as our autoregressive baseline.

Hyperparameters. For the hyperparameters mentioned in Section 3.2, we set $\lambda = 0.3$ to penalise slot orderings that achieve high immediate confidence and force subsequent slots into low-probability regions, guiding the search towards globally coherent generation trajectories. We also use $c = 50$, $N_{sim} = 256$, and $\tau = 0.5$. Detailed explanations of hyperparameter choice and sensitivity analysis are presented in Section 6 and Appendix C.2.

5. Experimental Results

Table 1 reports the performance of DIFFUSEARCH across six benchmarks, compared against both masked diffusion models (MDMs) and autoregressive models (ARMs). The results highlight three main phenomena:

1. DIFFUSEARCH outperforms both MDMs and ARMs baselines across five out of six benchmarks. *Additionally,* DIFFUSEARCH *significantly outperforms all MDMs across all six benchmarks.* For instance, DIFFUSEARCH achieves performance increases of 25.98% on HumanEval and 4.00% on MATH500 compared to ReFusion, with more substantial gains on traditional MDMs. (Li et al., 2025).

2. Coding tasks benefit more from MCTS slot planning. DIFFUSEARCH yields substantially larger gains on code-generation benchmarks, achieving improvements of 16.32% on HumanEval and 19.45% on MBPP. In contrast, improvements on multiple-choice reasoning benchmarks are more modest (ARC Challenge: 0.7%; GPQA: 4.33%). This pattern suggests that tasks involving structured program synthesis benefit more from strategic slot planning, as dependencies between code components—such as variable declarations, function definitions, and control flow—induce ordering constraints that can be effectively explored by MCTS. Further analysis in Appendix D.2 further confirms this hypothesis.

3. DIFFUSEARCH produces more compact and coherent reasoning. Under identical experimental conditions with chain-of-thought prompting, ARMs often produce lengthy reasoning sequences that may become incoherent or incomplete, particularly when the reasoning chain exceeds the model's effective context length on tasks, like GPQA-Diamond, that require extensive reasoning (see Appendix E for some qualitative examples). In contrast, DIFFUSEARCH generates more compact reasoning. For example, on MATH500 DIFFUSEARCH produces sequences of average length equal to 152.2 while Qwen2.5 7B has average prediction length equal to 436.0 while getting worse accuracy. The length reduction is consistent across the three benchmarks that require extensive reasoning (i.e., GSM8K, MATH500 and GPQA) and is always statistically significant ($p < 0.001$) as reported in Appendix D.1. We compare the performance of DIFFUSEARCH and autoregressive models at their maximum token lengths in Appendix D.3.

6. Analysis and Discussion

We provide a detailed analysis to better understand when and why MCTS-based slot planning improves generation performance. In particular, we examine how often non-sequential ordering decisions arise and when they matter, how exploration and simulation budget affect accuracy, and what computational trade-offs are introduced by search-based planning. Together, these analyses shed light on the

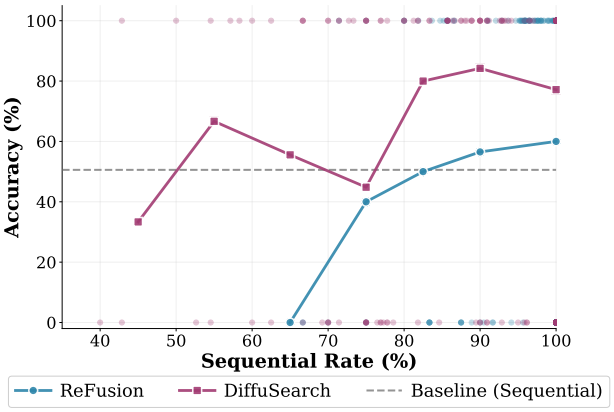


Figure 2. Relationship between generation sequentiality and accuracy on the MBPP dataset. Each dot represents a sample plotted by its accuracy and sequentiality rate. Darker dots denote higher density, reflecting multiple instances with identical sequentiality. Solid lines denote average accuracy trends computed by binning sequentiality rates for ReFusion and DIFFUSEARCH, while the dashed line indicates the overall accuracy of sequential (left-to-right) baseline.

practical behavior and efficiency of DIFFUSEARCH beyond aggregate benchmark results.

When Do Non-Sequential Slot Orderings Matter? We observe that the majority of slot-ordering decisions made by DIFFUSEARCH follow a sequential, left-to-right pattern, consistent with standard autoregressive decoding (91.1% for coding tasks and 93.8% for mathematical reasoning tasks). As shown in Figure 2, most instances concentrate at high sequentiality rates, indicating that the sequential order serves as an effective default strategy across tasks.

Despite their relative rarity, non-sequential decisions play a disproportionate role in performance improvements. Among instances where DIFFUSEARCH succeeds while the sequential autoregressive baseline fails (13.2% of the dataset), 60.7% involve at least one non-sequential ordering decision. This effect is also reflected in Figure 2, where higher accuracy is often associated with intermediate sequentiality rates rather than strictly sequential generation. Together, these observations suggest an effective exploration-exploitation trade-off: the search policy largely exploits the reliable sequential prior, while selectively deviating from it to resolve challenging constraints that confound greedy left-to-right decoding. This indicates that even sparse, well-timed departures from the sequential order can yield meaningful accuracy improvements when guided by informed slot planning. Further example and analysis are provided in Figure 4 and Appendix G.

How Do Exploration and Simulation Budget Affect Performance? We analyse the interaction between the exploration constant (c) and the simulation budget (N_{sim}) and their impact on the model's performance.

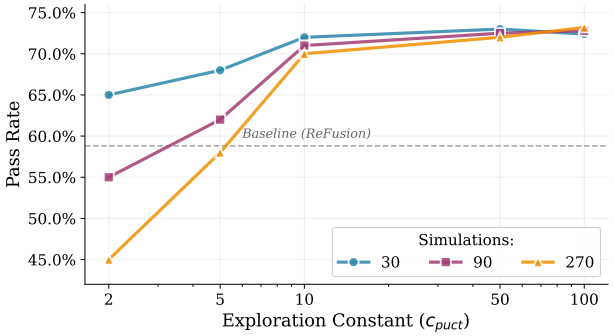


Figure 3. Impact of exploration constant (c) and simulation budget (N_{sim}) on task performance.

Config	Mean H (bits)	Std H	Median H	Concentration
Low Exploration ($c = 2$)				
$N = 30$	1.4062	0.3785	1.4824	0.4958
$N = 270$	1.1842	0.4043	1.1492	0.6118
High Exploration ($c = 100$)				
$N = 30$	1.4176	0.3771	1.5058	0.4954
$N = 270$	1.4340	0.3768	1.5211	0.4879

Table 2. Impact of simulation count (N_{sim}) and exploration constant (c) on policy entropy, H and concentration. Under low exploration $c = 2$, entropy decreases with more simulations, while under high exploration $c = 100$, entropy remains stable.

As shown in Figure 3 and Figure 8 in Appendix I, increasing the exploration budget c consistently improves accuracy across benchmarks, indicating that stronger exploration is necessary to overcome the local confidence bias induced by the baseline's predictions.

On the other hand, increasing the simulation budget (N_{sim}) does not guarantee an improvement in performance: at low exploration budget ($c = 2.0$) increasing N_{sim} from 30 to 270 simulations leads to a decrease in accuracy on both MBPP and MATH500. This phenomenon is explained by the entropy analysis in Table 2: under low exploration ($c = 2$), mean entropy drops from 1.41 to 1.18 bits as N_{sim} increases, indicating premature convergence towards locally confident but globally myopic orderings. Conversely, with high exploration ($c = 100$), entropy remains stable ≈ 1.43bits), allowing the search to maintain breadth and discover effective orderings. This suggests that, under insufficient exploration pressure, additional simulations reinforce early, locally confident but globally myopic slot orderings.

Regarding computational overhead, generation time scales approximately linearly with N_{sim} while varying c causes negligible cost as it only affects selection without requir-

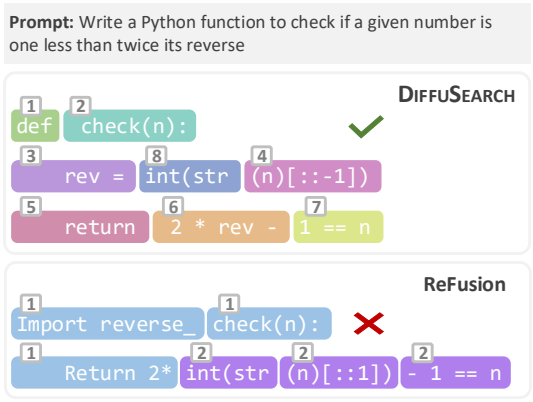


Figure 4. Comparison of ReFusion and DIFFUSEARCH on a coding prompt from MBPP. Superscripts denote the infilling slot order and colours indicate the specific generation step.

ing additional forward passes. We observe an accuracy-efficiency trade-off: although $N_{\text{sim}} = 270$ achieves highest accuracy, the improvement over $N_{\text{sim}} = 30 < 2\%$ while computational cost increases ninefold. A high-exploration, low-budget configuration achieves substantial gains such as 13.62% improvement on MBPP while maintaining reasonable inference time. (Detailed analysis and comparison with ReFusion can be found in Appendix H.)

These results highlight that the primary challenge in non-autoregressive slot planning is not search depth, but sufficient breadth to escape local optima.

How does MCTS mitigate local optima in slot-based generation? We give an intuition through the qualitative example in Figure 4. In the Figure, ReFusion selects and fills multiple high-confidence slots simultaneously from the start (slots 1, 2, and 3) in its attempt to maximise throughput. Specifically, it greedily fills slot 1 with an unnecessary import statement (i.e., "`Import reverse_`"), likely driven by a generic prior that Python scripts often begin with imports. However, because ReFusion cannot backtrack to correct this early commitment, this unnecessary slot forces the subsequent generation into a syntactically broken state (e.g., merging the import line directly into a part of a function definition header). On the other hand, DIFFUSEARCH avoids this issue through its lookahead capability. By simulating the full trajectory before committing, it detects that starting with an import leads to lower long-term coherence for this specific task, and instead correctly prioritises the function definition (by filling in slot 1 with "`def`" and then slot 2 with "`check(n):`" to complete the header of the function definition) as the most effective starting point.

7. Related work

Monte Carlo Tree Search. MCTS (Kocsis & Szepesvári, 2006; Coulom, 2006) has proven highly effective for sequential decision-making (Chaslot et al., 2008; Browne et al., 2012; van Krieken et al., 2025), achieving notable success in game-playing agents such as AlphaGo (Granter et al., 2017) and AlphaZero (Silver et al., 2017). Recently, MCTS has been extensively applied to enhance LLM Reasoning, including math (Gao et al., 2024; Xie et al., 2024; Wu et al., 2024), coding (Wang et al., 2025), and others. MCTS has also been applied to visual diffusion models (Yoon et al., 2025a;b; Ramesh & Mardani, 2025) for improving generation quality. To the best of our knowledge, our work represents the first application of MCTS to MDMs for text generation, specifically addressing slot ordering.

Masked Diffusion Models (MDMs). MDMs, originally successful in visual domains (Song et al., 2021; Ho et al., 2020), have emerged as a promising paradigm for non-autoregressive sequence generation (Nie et al., 2025; Bie et al., 2025; Gong et al., 2024), particular within the coding domain (Gong et al., 2025; Zhao et al., 2025). However, MDMs still struggle to match the performance of autoregressive models (ARMs) on complex reasoning tasks (Zhu et al., 2025). Architectures such as BD3LM (Arriola et al., 2025), Eso-LMs (Sahoo et al., 2025), and ReFusion (Li et al., 2025) have explored various decoding approaches to bridge this gap. Probabilistic methods such as top_k (Nie et al., 2025), low entropy (Ben-Hamu et al., 2025) and probability margins between top candidates (Kim et al., 2025) have been widely used in decoding within text MDMs.

8. Conclusion

We introduce DIFFUSEARCH, a training-free framework that enhances MDMs through strategic slot selection. By formulating slot ordering as decision-making and using MCTS with designated adaptations – confidence-aware value propagation and adaptive exploration budgets – DIFFUSEARCH systematically navigates the combinatorial space of generation orders without additional training. Through experiments across six reasoning benchmarks, DIFFUSEARCH demonstrates consistent improvements over existing MDM baselines and competitive performance with ARMs under identical experimental conditions, with particularly strong gains on code generation tasks. Our analysis reveals that (1) while DIFFUSEARCH mainly adheres to sequential ordering, strategic deviations to non-sequential generation are essential for maximising performance, and (2) rather than increasing the number of simulations, significant exploration breadth is needed to overcome the model's priors and discover effective slot orderings. Together, These findings establish DIFFUSEARCH as an effective approach for enhancing slot selection and infilling within MDMs.

Impact Statement

This paper presents work whose goal is to apply a controlled "plan-and-infill" search method using MCTS to masked diffusion models without requiring additional training, with applications in mathematical reasoning, coding, and general scientific reasoning tasks within language models. This could improve how diffusion models process generations by exploring multiple search paths with lookahead, reducing propagation of unmodelled dependencies across iterations that undermine global coherence. Our analysis on models requiring a combination of sequential and non-sequential generation strategies for maximised performance could be useful for future work on improving LLM reasoning, merging the realm between diffusion models and autoregressive models. Lastly, our results on large exploration constants being necessary to overcome the model's confidence priors could bring a societal impact towards the potential of structured "plan-and-infill" methods, or any diffusion-guided generation paradigms. There are potential societal consequences of our work, none of which we feel must be specifically highlighted here.

References

Arriola, M., Sahoo, S. S., Gokaslan, A., Yang, Z., Qi, Z., Han, J., Chiu, J. T., and Kuleshov, V. Block diffusion: Interpolating between autoregressive and diffusion language models. In International Conference on Learning Representations, 2025.

Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C., Terry, M., Le, Q., et al. Program synthesis with large language models. ArXiv preprint, abs/2108.07732, 2021.

Bellman, R. A markovian decision process. Indiana University Mathematics Journal, 6:679–684, 1957.

Ben-Hamu, H., Gat, I., Severo, D., Nolte, N., and Karrer, B. Accelerated sampling from masked diffusion models via entropy bounded unmasking. ArXiv preprint, abs/2505.24857, 2025.

Bie, T., Cao, M., Chen, K., Du, L., Gong, M., Gong, Z., Gu, Y., Hu, J., Huang, Z., Lan, Z., et al. Llada2. 0: Scaling up diffusion language models to 100b. ArXiv preprint, abs/2512.15745, 2025.

Browne, C. B., Powley, E., Whitehouse, D., Lucas, S. M., Cowling, P. I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., and Colton, S. A survey of monte carlo tree search methods. IEEE Transactions on Computational Intelligence and AI in games, 4(1):1–43, 2012.

Chaslot, G. M.-B., Winands, M. H., and van Den Herik, H. J. Parallel monte-carlo tree search. In International Conference on Computers and Games, pp. 60–71. Springer, 2008.

Chen, M. Evaluating large language models trained on code. ArXiv preprint, abs/2107.03374, 2021.

Clark, P., Cowhey, I., Etzioni, O., Khot, T., Sabharwal, A., Schoenick, C., and Tafjord, O. Think you have solved question answering? try arc, the ai2 reasoning challenge. ArXiv preprint, abs/1803.05457, 2018.

Cobbe, K., Kosaraju, V., Bavarian, M., Chen, M., Jun, H., Kaiser, L., Plappert, M., Tworek, J., Hilton, J., Nakano, R., Hesse, C., and Schulman, J. Training verifiers to solve math word problems. ArXiv preprint, abs/2110.14168, 2021.

Coulom, R. Efficient selectivity and backup operators in monte-carlo tree search. In International conference on computers and games, pp. 72–83. Springer, 2006.

Fu, H., Huang, B., Adams, V., Wang, C., Srinivasan, V., and Jiao, J. From bits to rounds: Parallel decoding with exploration for diffusion language models. ArXiv preprint, abs/2511.21103, 2025.

Gao, Z., Niu, B., He, X., Xu, H., Liu, H., Liu, A., Hu, X., and Wen, L. Interpretable contrastive monte carlo tree search reasoning. ArXiv preprint, abs/2410.01707, 2024.

Gong, S., Agarwal, S., Zhang, Y., Ye, J., Zheng, L., Li, M., An, C., Zhao, P., Bi, W., Han, J., et al. Scaling diffusion language models via adaptation from autoregressive models. ArXiv preprint, abs/2410.17891, 2024.

Gong, S., Zhang, R., Zheng, H., Gu, J., Jaitly, N., Kong, L., and Zhang, Y. Diffucoder: Understanding and improving masked diffusion models for code generation. ArXiv preprint, abs/2506.20639, 2025.

Granter, S. R., Beck, A. H., and Papke Jr, D. J. Alphago, deep learning, and the future of the human microscopist. Archives of pathology & laboratory medicine, 141(5): 619–621, 2017.

Guan, X., Zhang, L. L., Liu, Y., Shang, N., Sun, Y., Zhu, Y., Yang, F., and Yang, M. rstar-math: Small llms can master math reasoning with self-evolved deep thinking. ArXiv preprint, abs/2501.04519, 2025.

Hendrycks, D., Burns, C., Kadavath, S., Arora, A., Basart, S., Tang, E., Song, D., and Steinhardt, J. Measuring mathematical problem solving with the math dataset. NeurIPS, 2021.

Ho, J., Jain, A., and Abbeel, P. Denoising diffusion probabilistic models. In Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M., and Lin, H. (eds.), Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual, 2020.

Kim, J., Shah, K., Kontonis, V., Kakade, S., and Chen, S. Train for the worst, plan for the best: Understanding token ordering in masked diffusions. ArXiv preprint, abs/2502.06768, 2025.

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

Kojima, T., Gu, S. S., Reid, M., Matsuo, Y., and Iwasawa, Y. Large language models are zero-shot reasoners. In Koyejo, S., Mohamed, S., Agarwal, A., Belgrave, D., Cho, K., and Oh, A. (eds.), Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022, 2022.

Leang, J. O. J., Gema, A. P., and Cohen, S. B. Comat: Chain of mathematically annotated thought improves mathematical reasoning. In Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing, pp. 20256–20285, 2025a.

Leang, J. O. J., Zhao, Z., Gema, A. P., Yang, S., Kwan, W.-C., He, X., Li, W., Minervini, P., Giunchiglia, E., and Cohen, S. B. Picsar: Probabilistic confidence selection and ranking for reasoning chains. ArXiv preprint, abs/2508.21787, 2025b.

Li, J.-N., Guan, J., Wu, W., and Li, C. Refusion: A diffusion large language model with parallel autoregressive decoding. ArXiv preprint, abs/2512.13586, 2025.

Li, Y., Zhou, K., Zhao, W. X., and Wen, J. Diffusion models for non-autoregressive text generation: A survey. In Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence, IJCAI 2023, 19th-25th August 2023, Macao, SAR, China, pp. 6692–6701. ijcai.org, 2023. doi: 10.24963/IJCAI.2023/750.

Liu, A., Feng, B., Xue, B., Wang, B., Wu, B., Lu, C., Zhao, C., Deng, C., Zhang, C., Ruan, C., et al. Deepseek-v3 technical report. ArXiv preprint, abs/2412.19437, 2024.

Lyu, Q., Havaldar, S., Stein, A., Zhang, L., Rao, D., Wong, E., Apidianaki, M., and Callison-Burch, C. Faithful chain-of-thought reasoning. In Park, J. C., Arase, Y., Hu, B., Lu, W., Wijaya, D., Purwarianti, A., and Krisnadhi, A. A. (eds.), Proceedings of the 13th International Joint Conference on Natural Language Processing and the 3rd Conference of the Asia-Pacific Chapter of the Association for Computational Linguistics (Volume 1: Long Papers), pp. 305–329, Nusa Dua, Bali, 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.ijcnlp-main.20.

Nie, S., Zhu, F., You, Z., Zhang, X., Ou, J., Hu, J., Zhou, J., Lin, Y., Wen, J.-R., and Li, C. Large language diffusion models. ArXiv preprint, abs/2502.09992, 2025.

Prabhudesai, M., Chen, L., Ippoliti, A., Fragkiadaki, K., Liu, H., and Pathak, D. Maximizing confidence alone improves reasoning. ArXiv preprint, abs/2505.22660, 2025.

Ramesh, V. and Mardani, M. Test-time scaling of diffusion models via noise trajectory search. ArXiv preprint, abs/2506.03164, 2025.

Rein, D., Hou, B. L., Stickland, A. C., Petty, J., Pang, R. Y., Dirani, J., Michael, J., and Bowman, S. R. Gpqa: A graduate-level google-proof q&a benchmark. In First Conference on Language Modeling, 2024.

Sahoo, S. S., Yang, Z., Akhauri, Y., Liu, J., Singh, D., Cheng, Z., Liu, Z., Xing, E., Thickstun, J., and Vahdat, A. Esoteric language models. ArXiv preprint, abs/2506.01928, 2025.

Schrittwieser, J., Antonoglou, I., Hubert, T., Simonyan, K., Sifre, L., Schmitt, S., Guez, A., Lockhart, E., Hassabis, D., Graepel, T., et al. Mastering atari, go, chess and shogi by planning with a learned model. Nature, 588(7839): 604–609, 2020.

Schuster, T., Fisch, A., Gupta, J., Dehghani, M., Bahri, D., Tran, V., Tay, Y., and Metzler, D. Confident adaptive language modeling. In Koyejo, S., Mohamed, S., Agarwal, A., Belgrave, D., Cho, K., and Oh, A. (eds.), Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022, 2022.

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., Chen, Y., Lillicrap, T., Hui, F., Sifre, L., van den Driessche, G., Graepel, T., and Hassabis, D. Mastering the game of Go without human knowledge. Nature, 550(7676):354–359, 2017.

Song, Y., Sohl-Dickstein, J., Kingma, D. P., Kumar, A., Ermon, S., and Poole, B. Score-based generative modeling through stochastic differential equations. In 9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021. OpenReview.net, 2021.

Team, Q. et al. Qwen2 technical report. ArXiv preprint, abs/2407.10671, 2024.

van Krieken, E., Minervini, P., Ponti, E., and Vergari, A. Neurosymbolic diffusion models. ArXiv preprint, abs/2505.13138, 2025.

Wang, Y., Ji, P., Yang, C., Li, K., Hu, M., Li, J., and Sartoretti, G. Mcts-judge: Test-time scaling in llm-as-a-judge for code correctness evaluation. ArXiv preprint, abs/2502.12468, 2025.

Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., Chi, E. H., Le, Q. V., and Zhou, D. Chain-of-thought prompting elicits reasoning in large language models. In Koyejo, S., Mohamed, S., Agarwal, A., Belgrave, D., Cho, K., and Oh, A. (eds.), Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022, 2022.

Wu, J., Feng, M., Zhang, S., Che, F., Wen, Z., Liao, C., and Tao, J. Beyond examples: High-level automated reasoning paradigm in in-context learning via mcts. ArXiv preprint, abs/2411.18478, 2024.

Xie, Y., Goyal, A., Zheng, W., Kan, M.-Y., Lillicrap, T. P., Kawaguchi, K., and Shieh, M. Monte carlo tree search boosts reasoning via iterative preference learning. ArXiv preprint, abs/2405.00451, 2024.

Yang, A., Li, A., Yang, B., Zhang, B., Hui, B., Zheng, B., Yu, B., Gao, C., Huang, C., Lv, C., et al. Qwen3 technical report. ArXiv preprint, abs/2505.09388, 2025a.

Yang, W., Liao, M., and Fan, K. Markov chain of thought for efficient mathematical reasoning. In Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers), pp. 7132–7157, 2025b.

Ye, J., Xie, Z., Zheng, L., Gao, J., Wu, Z., Jiang, X., Li, Z., and Kong, L. Dream 7b: Diffusion large language models. ArXiv preprint, abs/2508.15487, 2025.

Yoon, J., Cho, H., Baek, D., Bengio, Y., and Ahn, S. Monte carlo tree diffusion for system 2 planning. ArXiv preprint, abs/2502.07202, 2025a.

Yoon, J., Cho, H., Bengio, Y., and Ahn, S. Fast monte carlo tree diffusion: 100x speedup via parallel sparse planning. ArXiv preprint, abs/2506.09498, 2025b.

Zhao, S., Gupta, D., Zheng, Q., and Grover, A. d1: Scaling reasoning in diffusion large language models via reinforcement learning. ArXiv preprint, abs/2504.12216, 2025.

Zhu, F., Wang, R., Nie, S., Zhang, X., Wu, C., Hu, J., Zhou, J., Chen, J., Lin, Y., Wen, J.-R., et al. Llada 1.5: Variance-reduced preference optimization for large language diffusion models. ArXiv preprint, abs/2505.19223, 2025.

A. Detailed DIFFUSEARCH algorithm

This section details the constituent phases of the DIFFUSEARCH algorithm. While the main text above outlines the stochastic confidence rollout used for value estimation (Algorithm 1), here we provide the complete pseudo-code for selecting the next slot.

In particular, Algorithm 2 outlines the overall slot selection framework. The specific subroutines are detailed in Algorithm 3 (SELECT), Algorithm 4 (EXPAND), and Algorithm 5 (BACKPROPAGATE). Note that, in Algorithm 2, the ROLLOUT function call refers directly to Algorithm 1. Additionally, the SELECTCHILD function applies the same PUCT criterion as Algorithm 3 (SELECT), but performs only a single selection step from the current node.

Algorithm 2 MCTS-based Slot Selection for DIFFUSEARCH

Input: Current state $s_t = (\sigma_t, \mathbf{x}_t)$, slot-level confidence scores $\{\mathcal{R}(s_t, a)\}_{a \in \mathcal{A}_t}$
Parameters: Number of simulations N_{sim}, exploration constant c, temperature τ, mixing coefficient λ
Output: Next action $a^* \in \mathcal{A}_t$ (slot index to fill next)
// Initialise root node and global statistics
$s_{\text{root}} \leftarrow s_t$
$N_{s_t} \leftarrow 0$ // Total visits to root state
for all $a \in \mathcal{A}_t$: $N^a_{s_t} \leftarrow 0, W(s_t, a) \leftarrow 0$
// Normalise confidence scores to obtain priors
$\{P(s_t, a)\}_{a \in \mathcal{A}_t} \leftarrow \text{Normalise}(\mathcal{R}(s_t, a)_{a \in \mathcal{A}_t})$
for $n \leftarrow 1$ **to** N_{sim} **do**
  // 1. Selection: traverse tree using PUCT to reach a leaf
  $s_{\text{leaf}} \leftarrow \text{SELECT}(s_{\text{root}}, c)$
  // 2. Expansion: if leaf is non-terminal and unexpanded, expand it
  if s_{leaf} is not terminal **and** s_{leaf} has no children **then**
    $\text{EXPAND}(s_{\text{leaf}}, \{\mathcal{R}(s_{\text{leaf}}, a)\})$ // Creates children, initialises stats
    $s_{\text{eval}} \leftarrow \text{SELECTCHILD}(s_{\text{leaf}}, c)$ // Pick one child
  else
    $s_{\text{eval}} \leftarrow s_{\text{leaf}}$
  end if
  // 3. Simulation: mix immediate confidence with rollout value
  $V \leftarrow \lambda \cdot \mathcal{R}(s_{\text{leaf}}, s_{\text{eval}}.\text{action}) + (1 - \lambda) \cdot \text{ROLLOUT}(s_{\text{eval}}, \tau, \{\mathcal{R}(s_{\text{eval}}, a)\})$ *// s_{eval}.action is the action that led to s_{eval}*
  // 4. Backpropagation: update statistics to root
  $\text{BACKPROPAGATE}(s_{\text{eval}}, V, s_{\text{root}})$
end for
// Select action with most visits
$a^* \leftarrow \arg\max_{a \in \mathcal{A}_t} N^a_{s_t}$
return a^*

Algorithm 3 SELECT

Input: Current state s, exploration constant c
Output: Selected leaf node
while s has children **do**
  // Calculate PUCT for all valid actions a from state s
  $a^* \leftarrow \arg\max_{a \in \mathcal{A}(s)} \left(Q(s, a) + c \cdot P(s, a) \frac{\sqrt{N_s}}{1 + N^a_s} \right)$
  $s \leftarrow \text{child}(s, a^*)$ *// Traverse to the child node corresponding to action a^**
end while
return s

B. Stochastic Confidence Rollout Example

As described in Section 3, we use stochastic confidence rollouts to estimate the long-term value of partial generation states during MCTS simulation. In particular, Algorithm 1 performs the temperature-scaled stochastic sampling over remaining unfilled slots, where slots with higher model confidence receive proportionally higher selection probability but are not deterministically chosen. This stochasticity enables exploration of diverse completion paths, capturing the model's uncertainty about optimal slot orderings whilst maintaining bias towards high-confidence regions of the generation space. This section provides the detailed algorithm and a concrete numerical example to illustrate the rollout mechanism.

Algorithm 4 EXPAND

Input: State s, slot-level confidence scores $\{\mathcal{R}(s,a)\}$
// Identify valid actions (indices of currently unfilled slots)
$\mathcal{A}_s \leftarrow \{\text{indices of unfilled slots in } s\}$
$R_{\text{total}} \leftarrow \sum_{a \in \mathcal{A}_s} \mathcal{R}(s,a)$
// Create a child node for each a and initialise its statistics
for each action $a \in \mathcal{A}_s$ **do**
  $N_s^a \leftarrow 0, \quad W(s,a) \leftarrow 0$
  $P(s,a) \leftarrow \mathcal{R}(s,a)/R_{\text{total}}$ *// Normalise to get prior probability*
  Create child node s' for action a and link as child(s,a)
end for

Algorithm 5 BACKPROPAGATE

Input: State s, values V estimated during rollout, root s_{root}
while $s \neq s_{\text{root}}$ **do**
  $p \leftarrow$ parent of s *// Get parent node of s*
  $a \leftarrow$ action from p to s *// Get action that led from* p to s
  // Update statistics for the edge (p,a)
  $N_p^a \leftarrow N_p^a + 1$
  $W(p,a) \leftarrow W(p,a) + V(p,a)$
  $Q(p,a) \leftarrow W(p,a)/N_p^a$
  // Update total visits for parent state
  $N_p \leftarrow N_p + 1$
  $s \leftarrow p$ *// Move up the tree*
end while

Consider a state where slots 0 and 1 are filled, and we must evaluate the path forward for slots $\{2,3,4\}$ with confidences $R(s,2) = 0.45$, $R(s,3) = 0.82$, $R(s,4) = 0.63$ and temperature $T = 0.5$.

Iteration 1: The rollout computes the sampling distribution:

$$p_2 = \frac{\exp(0.45/0.5)}{\sum} = \frac{2.460}{11.140} = 0.221$$
$$p_3 = \frac{\exp(0.82/0.5)}{\sum} = \frac{5.155}{11.140} = 0.463$$
$$p_4 = \frac{\exp(0.63/0.5)}{\sum} = \frac{3.525}{11.140} = 0.316$$

The algorithm samples $a^* = 3$ (higher confidence slot has higher probability but selection is stochastic). Accumulated reward: $V_{\text{total}} = 0.82$, $n = 1$.

Iteration 2: With remaining slots $\{2,4\}$:

$$p_2 = 0.411, \quad p_4 = 0.589$$

Sample $a^* = 4$. Update: $V_{\text{total}} = 1.45$, $n = 2$.

Iteration 3: Only slot 2 remains, select deterministically. Final: $V_{\text{total}} = 1.90$, $n = 3$.

Return: $V(s) = 1.90/3 = 0.633$, representing the expected quality of completing generation from this state.

This stochastic sampling enables the rollout to explore different completion paths, capturing the model's uncertainty about optimal slot ordering.

C. Experimental Setup

C.1. Additional Implementation Details

To ensure fair comparison across all methods, we maintain consistent hyperparameters throughout our experiments. All models use an evaluation length of 512 tokens following the configuration from (Li et al., 2025), with increasing token to

1024 for MBPP due to task complexity. We employ greedy decoding with temperature 0.0 across all settings, with zero-shot learning. For ReFusion (Li et al., 2025), we adopt the default configuration: slot size 8, 2 serial blocks, slot threshold 0.9, and token threshold 0.9. For DIFFUSEARCH, we configure 30 MCTS simulations, slot size 4, 32 serial blocks, slot threshold 0.5, token threshold 0.6, and exploration constant $c = 10.0$. Autoregressive baselines employ greedy decoding with temperature 0.0 and top_p=1.0. All evaluations are conducted on 4 NVIDIA H100 GPUs. We average all across 3 seeds for consistency, using Seed $= 21, 42, 84$.

C.2. Additional Hyperparameter Tuning

We further perform an additional hyperparameter tuning on the temperature, τ, and λ. However, due to computational cost, we are unable to find the optimal grid search using all four hyperparameters stated in Section 6. We take a hyperparameter tuning of c_{puct} of 50.0 with simulation of 256 to find the optimal budget. We perform the hyperparameter tuning on MATH-500 dataset.

Table 3. Grid Search Results for MATH500: Accuracy (%) by τ and λ

$\tau \setminus \lambda$	0.0	0.3	0.5	0.7	0.9	1.0
0.1	43.20	44.40	43.80	44.00	43.40	43.00
0.3	44.80	47.00	46.20	46.40	45.20	44.60
0.5	45.60	**48.20**	47.20	47.40	45.80	44.80
0.7	45.00	47.20	46.40	46.60	45.40	44.20
1.0	43.60	44.80	44.20	44.20	43.60	43.00

Based on the table, we can observe that the best hyperparameter tuning lies on τ = 0.5, and $\lambda = 0.3$. This is because strikes a critical balance in the exploration-exploitation trade-off: it introduces sufficient stochasticity to prevent the search from converging prematurely on local optima (a failure mode observed at lower temperatures), while maintaining enough coherence to avoid the semantic degeneration typical of higher entropy distributions.

Furthermore, $\lambda = 0.3$ indicates that the value estimation benefits significantly from prioritizing long-term rollout feedback over immediate confidence. In our formulation, a lower λ shifts the weight towards the Monte Carlo rollout return rather than the immediate policy prior. This suggests that for complex mathematical reasoning, the verifiability of a full solution path (the rollout) is a more reliable signal than the model's local token probabilities, though the immediate confidence remains necessary as a regularizing prior to guide the initial search direction.

D. Experimental Results and Tests

D.1. Analysis between token length and accuracy

Table 4. Token efficiency and accuracy comparison between DIFFUSEARCH and Qwen2.5-7B across benchmark datasets. All reductions are statistically significant ($p < 0.001$).

	Token Count			Accuracy (%)		
Dataset	DIFFUSEARCH	Qwen2.5	Reduction	DIFFUSEARCH	Qwen2.5	Improvement
MBPP	44.0	368.2	−88.05%	72.43	64.04	+8.39
ARC	91.3	332.8	−72.55%	88.40	87.88	+0.52
MATH500	152.2	436.0	−65.08%	48.40	46.00	+2.40
GSM8K	184.0	276.2	−33.39%	88.63	86.43	+2.20
GPQA	34.6	534.3	−93.53%	34.78%	16.94%	+17.84
HumanEval	63.6	256.3	−75.19%	78.37%	71.78%	+6.59

Table 4 demonstrates that DIFFUSEARCH achieves a significant improvements in both token efficiency and accuracy compared to the autoregressive baseline. Across all benchmarks, DIFFUSEARCH reduces token consumption by an average of 64.77% ($p < 0.001$) while simultaneously improving accuracy by 3.38% percentage points. The efficiency gains are particularly great on code generation tasks, with MBPP exhibiting an 88.05% token reduction alongside an 8.39%

improvement. This pattern suggests that DIFFUSEARCH is especially effective at identifying compact reasoning paths for structured generation tasks. Even on datasets where accuracy improvements are modest (e.g., ARC with $+0.52\%$), the token reduction remains substantial at 72.55%, indicating that DIFFUSEARCH consistently produces more concise solutions without sacrificing correctness. The inverse relationship between token reduction and task complexity is evident: while MBPP and ARC show the largest reductions (88.05% and 72.55% respectively), GSM8K exhibits a more moderate 33.39% reduction, likely due to the sequential nature of multi-step arithmetic reasoning. Nevertheless, the consistent improvements across all benchmarks validate that MCTS-guided slot selection enables more efficient exploration of the solution space compared to purely confidence-based ordering. Figure 5 provides a clear visualisation of the token reduction.

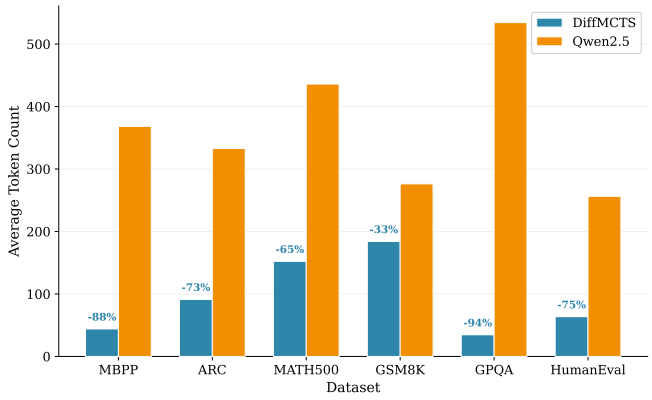


Figure 5. Comparison of token reduction across models. We observe that DIFFUSEARCH significantly reduces tokens while generating responses, demonstrating the compactness and coherence of using DIFFUSEARCH.

D.2. Task-Specific Improvements

To investigate whether MCTS-based slot planning benefits different task types differentially, we analyse the absolute accuracy improvements from ReFusion to DIFFUSEARCH across our evaluation benchmarks. Coding tasks (MBPP, HumanEval) demonstrate substantially larger improvements compared to reasoning and multiple-choice tasks. As shown in Table 5, this difference is statistically significant ($t = 2.97$, $p = 0.041$, Cohen's $d = 3.21$), with coding tasks exhibiting $3.13\times$ higher improvements. *We observe similar behaviour with Qwen3, given its similar architecture to Qwen2.5.* This suggests that MCTS-based planning particularly benefits domains with strict structural dependencies, such as variable declarations, function definitions, and control flow sequences, where slot ordering critically affects semantic validity.

Table 5. Statistical comparison of MCTS improvements between coding and non-coding tasks.

Metric	**Values**
Difference	7.43% (3.13×)
t-statistic	2.97
p-value	0.041
Cohen's d	3.21
Significance	$p < 0.05$ ✓

D.3. Impact of Extended Token Length on Autoregressive Models

We compare the total completion token lengths between autoregressive models and DIFFUSEARCH. Specifically, we evaluate Qwen2.5 7B, Qwen3 8B, and DIFFUSEARCH by extending their maximum token lengths to match their respective baselines: 32,768 tokens for both Qwen2.5 and Qwen3, and 1,024 tokens for ReFusion. Due to the nature of ReFusion and other existing MDMs (Nie et al., 2025; Zhao et al., 2025; Zhu et al., 2025; Li et al., 2025; Gong et al., 2025), the model's reasoning capabilities could not be extended for a fair comparison; however, we acknowledge that investigating token length extensions for diffusion-based models represents a promising direction for enhancing LLM reasoning capabilities. We restrict our analysis to MATH500, GSM8K, and GPQA-Diamond, as these are the only benchmarks mentioned in Section 4 with incomplete baseline results.

Table 6. Comparison of ReFusion-MCTS, Qwen2.5, and Qwen3 benchmarks.

Model	Context	GSM8K	MATH500	GPQA
DIFFUSEARCH	1024	**90.63%**	54.60%	37.19%
Llama3.1 7B	8096	84.91%	39.40%	37.71%
Qwen2.5 7B	32768	89.99%	63.40%	35.94%
Qwen3 8B	32768	88.17%	**68.00%**	**41.96%**

Table 6 demonstrates that DIFFUSEARCH achieves competitive performance while using significantly fewer tokens (1024 vs 32,768). Specifically,DIFFUSEARCH marginally outperforms both Qwen2.5 7B and Qwen3 8B on GSM8K. On GPQA,DIFFUSEARCH surpasses Qwen2.5 but underperforms relative to Qwen3 8B. However, DIFFUSEARCH exhibits weaker performance on MATH500, a benchmark requiring complex mathematical reasoning. Extending ReFusion to 4096 tokens leads to the same performance. In this case, we attribute this limitation to the inherent constraint of ReFusion and existing diffusion models, which cannot generate sequences beyond their predetermined token limit. We hypothesise that extending the maximum token length of diffusion-based models could substantially improve their reasoning capabilities.

E. Example of ReFusion generation length correspond to autoregressive

E.1. Example 1

DiffMCTS Generation *(Correct, 278 tokens)*

Question: A merchant wants to make a choice of purchase between 2 purchase plans: jewelry worth $5,000 or electronic gadgets worth $8,000. His financial advisor speculates that the jewelry market will go up 2.5% while the electronic gadgets market will rise 1.2% within the same month. If the merchant is looking to maximise profit at the end of this month by making a choice, how much profit would this be?
Ground Truth: 125

Generated Solution:
Step 1: Calculate the future value of the jewelry purchase.
Future value = Current value + (Current value $\times$ Percentage increase)
Future value = $5,000 + ($5,000 $\times$ 0.025) = $5,000 + $125 = $5,125
Step 2: Calculate the future value of the electronic gadgets purchase.
Future value = $8,000 + ($8,000 $\times$ 0.012) = $8,000 + $96 = $8,096
Step 3: Calculate the profit for each purchase plan.
For the jewelry purchase: Profit = $5,125 - $5,000 = $125
For the electronic gadgets purchase: Profit = $8,096 - $8,000 = $96
Step 4: Compare the profits and choose the purchase plan with the highest profit.
The jewelry purchase has a higher profit of $125 compared to the electronic gadgets purchase with a profit of $96.
The final answer is: $125
Answer: 125

Qwen2.5-7B Generation *TRUNCATED (512 tokens)*

Generated Solution (Truncated):
Step 1: Calculate the value of the jewelry after the market increase

- Initial value of jewelry: $5,000
- Speculated market increase: 2.5%

First, convert the percentage increase to a decimal: $2.5\% = \frac{2.5}{100} = 0.025$
Next, calculate the increase in value: Increase in value $= 5000 \times 0.025 = 125$
Finally, add the increase to the initial value: Final value of jewelry $= 5000 + 125 = 5125$
Step 2: Calculate the value of the electronic gadgets after the market increase

- Initial value of electronic gadgets: $8,000
- Speculated market increase: 1.2%

First, convert the percentage increase to a decimal: $1.2\% = \frac{1.2}{100} = 0.012$
Next, calculate the increase in value: Increase in value $= 8000 \times 0.012 = 96$
Finally, add the increase to the initial value: Final value of electronic gadgets $= 8000 + 96 = 8096$
Step 3: Determine the profit for each purchase
Profit from jewelry: Profit $= 5125 - 5000 = 125$
Profit from electronic gadgets: Profit $= 8096 - 8000 = 96$
Step 4: Compare the profits
Since $125 $>$ $96, the merchant should choose to purchase the jewelry to **[TRUNCATED]**

E.2. Example 2

DiffMCTS Generation *(Correct, 319 tokens)*

Question: Which of the following actions will increase the gravitational force between two masses, m_1 and m_2, by the greatest amount?
Ground Truth: C

Generated Solution:
The gravitational force between two masses is given by the formula $F = G \cdot \frac{m_1 \cdot m_2}{r^2}$, where G is the gravitational constant, m_1 and m_2 are the masses, and r is the distance between the masses. To increase the gravitational force, we need to increase the numerator $(m_1 \cdot m_2)$ and decrease the denominator (r^2).
Option (A): Halving the mass of m_1 and halving the distance between m_1 and m_2 will decrease the numerator by half and decrease the denominator by a quarter, resulting in a net decrease in the gravitational force.
Option (B): Halving the mass of m_1 and doubling the distance between m_1 and m_2 will decrease the numerator by half and increase the denominator by a quarter, resulting in a net decrease in the gravitational force.
Option (C): Doubling the mass of m_1 and halving the distance between m_1 and m_2 will increase the numerator by a factor of 2 and decrease the denominator by a quarter, resulting in a net increase in the gravitational force.
Option (D): Doubling the mass of m_1 and doubling the distance between m_1 and m_2 will increase the numerator by a factor of 2 and increase the denominator by a quarter, resulting in a net increase in the gravitational force.
Comparing the options, we can see that option (C) will result in the greatest increase in the gravitational force, as it increases the numerator by a factor of 2 and decreases the denominator by a quarter.
The final answer is: C
Answer: C

Qwen2.5-7B Generation *TRUNCATED (512 tokens)*

Generated Solution (Truncated):
To determine which action will increase the gravitational force between two masses, m_1 and m_2, by the greatest amount, we need to consider the formula for gravitational force:

$$F = G\frac{m_1 m_2}{r^2}$$

where F is the gravitational force, G is the gravitational constant, m_1 and m_2 are the masses, and r is the distance between the centres of the two masses.
Let's analyse each option step by step:
Option (A): Halve the mass of m_1 and halve the distance between m_1 and m_2.
New force:

$$F' = G\frac{\frac{m_1}{2} \cdot m_2}{\left(\frac{r}{2}\right)^2} = G\frac{\frac{m_1 m_2}{2}}{\frac{r^2}{4}} = G\frac{m_1 m_2}{2} \cdot \frac{4}{r^2} = 2 \cdot G\frac{m_1 m_2}{r^2} = 2F$$

The force is doubled.
Option (B): Halve the mass of m_1 and double the distance between m_1 and m_2.
New force:

$$F' = G\frac{\frac{m_1}{2} \cdot m_2}{(2r)^2} = G\frac{\frac{m_1 m_2}{2}}{4r^2} = \frac{1}{8} \cdot G\frac{m_1 m_2}{r^2} = \frac{1}{8}F$$

The force is reduced to one-eighth.
Option (C): Double the mass of m_1 and halve the distance between m_1 and m_2.
New force:

$$F' = G\frac{2m_1 \cdot m_2}{\left(\frac{r}{2}\right)^2} = G\frac{2m_1 m_2}{\frac{r^2}{4}} = 8 \cdot G\frac{m_1 m_2}{\text{[TRUNCATED]}}$$

E.3. Example 3

DiffMCTS Generation *(Correct, 403 tokens)*

Question: Find all values of x that satisfy the equation $x = \sqrt{11 - 2x} + 4$.
Ground Truth: $x = 5$

Generated Solution:
To solve the equation $x = \sqrt{11 - 2x} + 4$, we will first isolate the square root term and then square both sides of the equation to eliminate the square root.
Step 1: Isolate the square root term

$$x = \sqrt{11 - 2x} + 4 \Rightarrow x - 4 = \sqrt{11 - 2x}$$

Step 2: Square both sides of the equation

$$(x - 4)^2 = \left(\sqrt{11 - 2x}\right)^2 \Rightarrow x^2 - 8x + 16 = 11 - 2x$$

Step 3: Rearrange the equation to form a quadratic equation

$$x^2 - 8x + 16 - 11 + 2x = 0 \Rightarrow x^2 - 6x + 5 = 0$$

Step 4: Solve the quadratic equation

$$x^2 - 6x + 5 = 0 \Rightarrow (x - 1)(x - 5) = 0 \Rightarrow x = 1 \text{ or } x = 5$$

Step 5: Check the solutions in the original equation

$$x = 1: \quad 1 = \sqrt{11 - 2(1)} + 4 \Rightarrow 1 = \sqrt{9} + 4 \Rightarrow 1 = 3 + 4 \Rightarrow 1 = 7 \text{ (False)}$$

$$x = 5: \quad 5 = \sqrt{11 - 2(5)} + 4 \Rightarrow 5 = \sqrt{1} + 4 \Rightarrow 5 = 1 + 4 \Rightarrow 5 = 5 \text{ (True)}$$

The final answer is: $\boxed{5}$
Answer: 5

Qwen2.5-7B Generation *TRUNCATED (512 tokens)*

Generated Solution (Truncated):
To solve the equation $x = \sqrt{11 - 2x} + 4$, we will follow these steps:
1. Isolate the square root term:

$$x - 4 = \sqrt{11 - 2x}$$

2. Square both sides to eliminate the square root:

$$(x - 4)^2 = (\sqrt{11 - 2x})^2$$

Simplifying both sides:

$$(x - 4)^2 = 11 - 2x$$

3. Expand the left-hand side:

$$x^2 - 8x + 16 = 11 - 2x$$

4. Move all terms to one side to form a quadratic equation:

$$x^2 - 8x + 16 - 11 + 2x = 0$$

Simplifying:

$$x^2 - 6x + 5 = 0$$

5. Factor the quadratic equation:

$$(x - 1)(x - 5) = 0$$

6. Solve for x:

$$x - 1 = 0 \quad \text{or} \quad x - 5 = 0$$

$$x = 1 \quad \text{or} \quad x = 5$$

7. Verify the solutions by substituting them back into the original equation:
For $x = 1$:

$$1 = \sqrt{11 - 2(1)} + 4 \Rightarrow 1 = \sqrt{9} + 4 \Rightarrow 1 = 3 + 4 \Rightarrow 1 \neq 7$$

So, $x = 1$ is not a solution.
For $x = 5$:

$$5 = \sqrt{11 - 2(5)} + 4 \Rightarrow 5 = \sqrt{1} + 4$$ **[TRUNCATED]**

F. Complete MCTS Example Walkthrough

F.1. Problem Setup

We demonstrate the MCTS algorithm on a MATH500 problem with 4 remaining slots to fill. The model provides slot confidences: $R(s,0) = 0.35$, $R(s,1) = 0.78$, $R(s,2) = 0.52$, $R(s,3) = 0.41$.

F.2. Simulation 1: Initial Exploration

Selection: Start at ROOT (leaf node, no children).

Expansion: Compute priors by normalizing confidences:

$$P(s,a) = \frac{R(s,a)}{\sum_{a'} R(s,a')}$$
$$P(s,0) = 0.35/2.06 = 0.170, \quad P(s,1) = 0.379,$$
$$P(s,2) = 0.252, \quad P(s,3) = 0.199$$

Create 4 children. Select child with highest PUCT:

$$\text{PUCT}(a) = Q(s,a) + c \cdot P(s,a) \cdot \frac{\sqrt{N(s)}}{1 + N(s,a)}$$
$$\text{PUCT}(\text{slot } 1) = 0 + 1.4 \times 0.379 \times \frac{\sqrt{1}}{1+0} = 0.531 \quad \text{(max)}$$

Simulation: Perform stochastic rollout from state with slot 1 filled:

- Immediate reward: $r_1 = 0.78$
- Rollout: Sample slots $\{2, 3, 0\}$ stochastically, collect rewards $\{0.52, 0.41, 0.35\}$
- Rollout value: $V_{\text{rollout}} = (0.52 + 0.41 + 0.35)/3 = 0.427$
- Combined: $V = 0.3 \times 0.78 + 0.7 \times 0.427 = 0.533$

Backpropagation: Update Child_1 and ROOT:

$$N(\text{Child}_1) = 1, \quad Q(\text{Child}_1) = 0.533$$

F.3. Simulation 2: Deepening the Tree

Selection: Recompute PUCT at ROOT with $N(\text{ROOT}) = 1$:

$$\text{PUCT}(\text{slot } 0) = 0 + 1.4 \times 0.170 \times \frac{\sqrt{2}}{1} = 0.337$$
$$\text{PUCT}(\text{slot } 1) = 0.533 + 1.4 \times 0.379 \times \frac{\sqrt{2}}{2} = 0.909 \quad \text{(max)}$$

Select Child_1 again (highest PUCT despite 1 visit).

Expansion: Child_1 is now a leaf with filled slots $\{1\}$. Expand with remaining slots $\{0, 2, 3\}$, creating 3 grandchildren.

Simulation & Backpropagation: Select $\text{Child}_{1,2}$ (slot 2 after slot 1), perform rollout, get reward 0.422. Backpropagate to $\text{Child}_{1,2}$, Child_1, and ROOT.

F.4. Final Selection

After 4 simulations, the tree has visit counts:

$$\text{Visits} = \{\text{slot } 0 : 1,\ \text{slot } 1 : 2,\ \text{slot } 2 : 1,\ \text{slot } 3 : 0\}$$

Decision: Select slot 1 (most visits).

F.5. Key Observation

Although slot 1 had the highest initial confidence (0.78), MCTS confirmed this choice through *exploration*: rollouts from state $\{1\}$ consistently found high-value completion paths, increasing confidence that slot 1 is the optimal first choice.

G. Analysis between Sequence Rate and accuracy

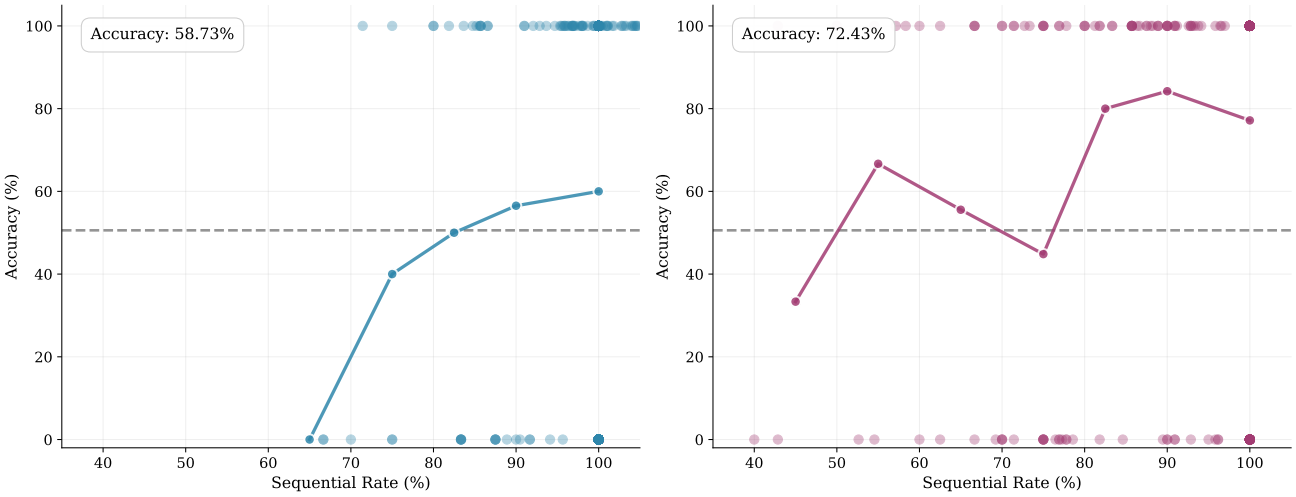
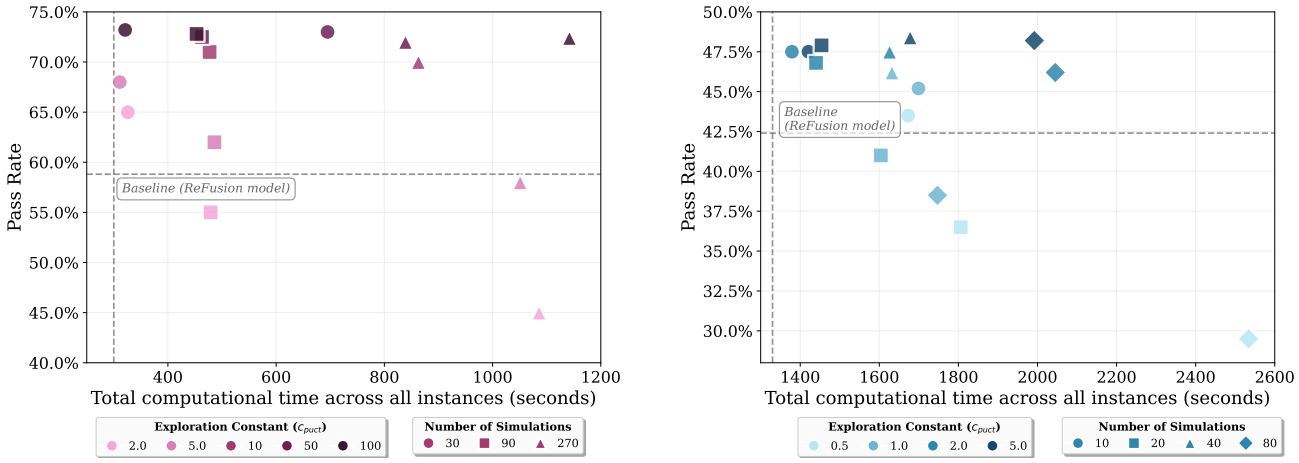


Figure 6. Plot with x axis being the sequential rate and y axis being the accuracy. ReFusion exhibits a positive correlation between sequential rate and accuracy, whilst DIFFUSEARCH achieves optimal performance at approximately 90% of sequential generation. This demonstrates that while predominantly sequential ordering is beneficial, strategic non-sequential generation leads to improved performance, mirroring human behaviour in coding tasks.

Figure 6 demonstrates a comparative analysis of sequential slot selection behaviour and its relationship to accuracy on the MBPP code generation benchmark for both ReFusion (baseline) and DIFFUSEARCH. The baseline ReFusion method (left panel, 58.81% accuracy) shows a weak positive correlation between sequential ordering and task success. In contrast, DIFFUSEARCH (right panel, 72.43% accuracy) demonstrates a more strategic approach while achieving substantially higher accuracy. The trend lines reveal that both methods benefit from sequential structure to some degree, consistent with code generation's inherent ordering dependencies (e.g., variable declarations preceding usage); however, MCTS's selective deviation from strict left-to-right generation (8.2% non-sequential exploration) enables it to avoid error propagation and handle complex logical dependencies more effectively. This strategic flexibility, rather than rigid adherence to sequential ordering, accounts for the performance gain, validating the hypothesis that optimal slot planning must adaptively balance sequential structure with non-sequential exploration.

H. More Efficiency Analysis of DIFFUSEARCH

In this section, we further answer the question: **what is the overhead caused by MCTS-Based Slot Planning?**. We compare the generation time of ReFusion with and without MCTS-based slot planning. As shown in Figure 7 and Figure 7a in Appendix H, generation time scales approximately linearly with the number of simulations N_{sim}. In contrast, varying the exploration constant c causes negligible overhead, as it only affects the selection criterion without requiring additional forward passes. This observation suggests that as expected computational cost is controlled primarily through the simulation budget.



(a) MBPP coding tasks

(b) MATH500 mathematical tasks

Figure 7. Accuracy versus generation time across different hyperparameter settings on coding and mathematical reasoning tasks.

The data reveals an accuracy-efficiency trade-off. While the highest accuracy is achieved with $N_{\text{sim}} = 270$, the improvement over $N_{\text{sim}} = 30$ is marginal (typically $< 2\%$), whereas the computational cost increases ninefold. This supports our earlier finding that simulation budgets suffice when exploration is adequate. Although MCTS introduces additional overhead compared to the baseline ReFusion inference, a high-exploration, low-budget configuration (e.g., $c = 100$, $N_{\text{sim}} = 30$) achieves substantial accuracy gains, such as a 13.62% improvement on MBPP, while maintaining reasonable inference time. Hardware details are provided in in Appendix C.1. In addition to the analysis of coding tasks as show in Figure 7b, we present the analysis using mathematical task MATH500 in Figure 7a. The results show a similar trend: there is a clear linear relationship between the number of simulations N_{sim} and computational time, while the exploration constant c incurs negligible overhead.

I. More Analysis on The Impact of Exploration and Simulation Budget

In addition to the analysis of coding tasks as shown in Figure 2, we provide the MATH500 mathematical reasoning task in Figure 8. We found a similar trend with coding tasks, where a higher exploration constant obtains higher accuracy.

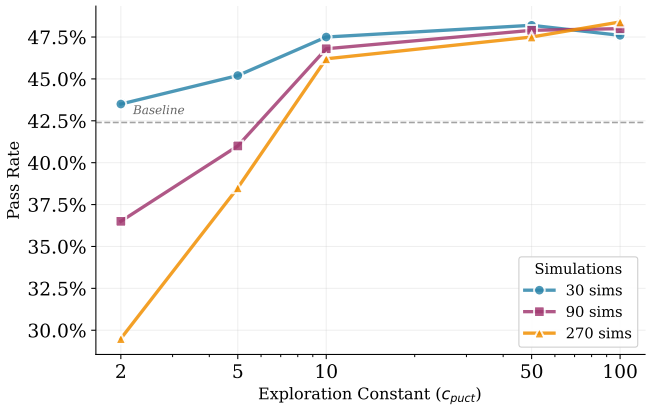


Figure 8. Impact of exploration constant (c_{puct}) and simulation budget on task performance on MATH500 mathematical reasoning tasks.