Accelerating Speculative Decoding with Block Diffusion Draft Trees

Liran Ringel[1] Yaniv Romano[1,2]

Abstract

Speculative decoding accelerates autoregressive language models by using a lightweight drafter to propose multiple future tokens, which the target model then verifies in parallel. DFlash shows that a block diffusion drafter can generate an entire draft block in a single forward pass and achieve state-of-the-art speculative decoding performance, outperforming strong autoregressive drafters such as EAGLE-3. Vanilla DFlash, however, still verifies only a single drafted trajectory per round, potentially limiting its acceptance length. We introduce DDTree (Diffusion Draft Tree), a method that constructs a draft tree directly from the per-position distributions of a block diffusion drafter. Under a fixed node budget, DDTree uses a simple best-first heap algorithm to select the continuations that are most likely to match the target model according to a surrogate defined by the draft model's output. The resulting tree is verified efficiently in a single target model forward pass using an ancestor-only attention mask. Because DDTree builds on DFlash, a leading draft model for speculative decoding, these gains place DDTree among the leading approaches to speculative decoding.

Project Page Code

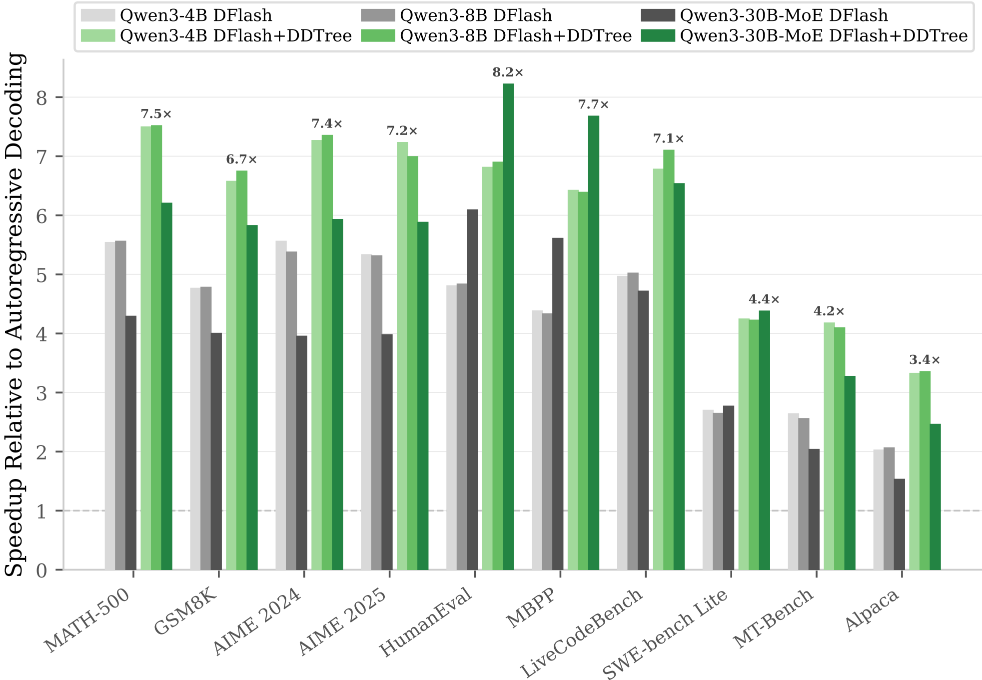


Figure 1: Speedups relative to autoregressive decoding at temperature 0.0 across datasets and target models. DDTree bars use the best tree-node budget for each dataset-model pair.

[1]Department of Computer Science, Technion – Israel Institute of Technology. [2]Department of Electrical and Computer Engineering, Technion – Israel Institute of Technology. Correspondence to: Liran Ringel <liranringel@cs.technion.ac.il>.

1 Introduction

Autoregressive language models generate text one token at a time, so the sampling of each new token requires another forward pass through a large model. This sequential dependence makes decoding a major source of inference latency. Speculative decoding addresses this bottleneck by using a lightweight drafter model to propose several future tokens and a large target model to verify them in parallel, while preserving the target model's output distribution [5, 13]. As language models continue to grow in size, reducing decoding latency without changing model outputs has made speculative decoding an increasingly important technique for inference.

The effectiveness of speculative decoding depends on the quality of the draft model. To deliver end-to-end speedups, the drafter must be cheap enough that drafting overhead stays small and accurate enough that the target model frequently accepts the drafted tokens. At a high level, the objective is to maximize the expected number of drafted tokens that the target model accepts, i.e., tokens that agree with the target model and can therefore be committed. At the same time, this should be achieved without adding significant overhead that makes the speedup disappear.

Block diffusion [2] is especially attractive in this setting as it can generate an entire draft block in a single forward pass. DFlash [6] shows the promise of this approach: it uses a small block diffusion drafter that leverages features derived from the larger target model, achieving state-of-the-art speculative decoding performance. Indeed, DFlash has been shown to outperform strong autoregressive drafters such as EAGLE-3 [16]. These results establish block diffusion as a powerful foundation for speculative decoding. At the same time, these promising results also spark a key challenge: how should we make the best use of the information that the block diffusion draft model produces?

Currently, DFlash verifies only one drafted trajectory per round, even though a single block diffusion forward pass produces a distribution over tokens at each future position. As such, DFlash does not utilize the many plausible continuations that block diffusion produces. Naturally, exploring more continuations could increase the probability that the target model continues along a drafted path. On the other hand, naively using multiple continuations increases the verifier cost and can erase the latency benefit. The challenge, therefore, is to use the draft model's per-position distributions to choose the continuations that are most worthwhile to verify.

We address this challenge with DDTree (Diffusion Draft Tree). Our method (i) constructs a draft tree directly from the per-position distributions produced by a block diffusion drafter, (ii) selects a compact set of promising continuations under a specified tree-node budget, and (iii) verifies them in a single target model forward pass with tree attention. In this way, DDTree preserves DFlash's low drafting latency while allowing each round to explore multiple continuations instead of only one. Our experiments show that DDTree consistently improves over vanilla DFlash across models, and Figure 1 summarizes these gains.

Contributions.

- We introduce DDTree, a speculative decoding method that constructs a draft tree directly from the per-position distributions produced by a single block diffusion forward pass within a fixed node budget.

- We show that the tree we construct provably maximizes the expected number of accepted tokens under the draft model, which we use as a surrogate for the target model's expected acceptance length. We also show that the optimal tree can be recovered with a simple and efficient best-first heap algorithm. This result builds on OPT-Tree [22], but adapts it to the block diffusion setting,

where the tree is constructed from the per-position distributions of a single forward pass rather than the multiple forward passes required by autoregressive drafters.

- We implement DDTree on top of the Hugging Face Transformers library and show consistent gains over vanilla DFlash across model sizes and domains.

2 Related Work

Speculative decoding and tree-based verification. Foundational speculative decoding methods verify a drafted continuation against the target model in parallel and preserve the target model's output distribution [5, 13]. A subsequent line of work generalizes this setting from a single drafted continuation to a tree of candidate continuations [11, 20, 23]. SpecInfer by Miao et al. [19] introduces tree attention for efficient target-model verification over token trees, and Medusa [4] combines multiple prediction heads with the same style of tree-based verification. The EAGLE family extends this direction using target-model features: EAGLE drafts in feature space [14], EAGLE-2 adds dynamic draft-tree construction [15], and EAGLE-3 predicts tokens directly from fused multi-layer features [16]. OPT-Tree by Wang et al. [22] constructs adaptive trees for autoregressive drafters by maximizing an approximate expected acceptance length under a node budget. In that autoregressive setting, tree construction still requires one drafter's forward pass per tree depth (i.e., per token position), and thus it has higher computational overhead compared to our DDTree approach.

Parallel drafting and block diffusion drafters. A complementary line of work reduces drafting latency by predicting multiple future tokens in a single forward pass. Block Diffusion provides the modeling foundation for prefix-conditioned blockwise denoising with KV-cache-friendly generation [2]. PARD [1] studies low-cost parallel drafting by adapting autoregressive models to mimic diffusion-style block prediction. DFlash [6] shows that a small block diffusion drafter, conditioned on target-model features, can predict an entire block in one forward pass and then verify a single drafted trajectory losslessly.

Vanilla DFlash, however, explores only one continuation per round. A very recent work, DART [18], also constructs draft trees from one-pass parallel logits. Still, it relies on continuity-aware tree pruning with an external N-gram continuity score and a large N-gram trie at runtime [18]. By contrast, our proposed DDTree keeps the same one-pass DFlash drafter and constructs the tree directly from the per-position probabilities produced by that pass. This avoids auxiliary external scoring and gives an explicit surrogate objective that our best-first construction provably maximizes.

3 Background: Block Diffusion Drafting

Speculative decoding methods proceed in rounds. At the beginning of each round, we already have one new token produced by the target model; we denote this 'bonus' token by b. This token is guaranteed to exist, even if no drafted tokens were accepted in the previous round. Importantly, although b has been selected by the target model, the target model has not yet been run on b, i.e., no forward pass has been performed with b appended to the context. As a result, the drafter can condition on the identity of b, but any target-model features it uses, as in DFlash, are only available for the preceding context.

With b fixed, the drafter's task is to predict the next tokens after b. A block diffusion drafter does this in parallel, producing a short block of future tokens rather than generating them autoregressively one token at a time. Given the context and b, the drafter takes a masked block of the form $[b, m, \dots, m]$ and predicts tokens for the next L masked positions in one forward pass. This produces

logits

$$\ell_i \in \mathbb{R}^{|\mathcal{V}|}, \qquad i = 1, \dots, L,$$

or, equivalently, per-position token distributions

$$q_i(v) = \mathrm{softmax}(\ell_i)_v, \qquad v \in \mathcal{V}.$$

Crucially, each q_i is a *marginal* distribution for position i, not a path-conditioned distribution. Because the block is predicted in parallel, the prediction at position i conditions on the context before the drafted block (and, in DFlash, on target-model features), but not on the specific token choices at positions $1, \dots, i-1$ inside that same block. Thus, a single DFlash pass provides one marginal distribution per future position, rather than conditional probabilities for every partial continuation.

This distinction is central to DDTree. Let c denote the current context, let b denote the bonus token available at the start of the round, and let $y_{1:L}$ denote a candidate continuation after b. Under the target model, the continuation distribution factorizes autoregressively as

$$p(y_{1:L} \mid c, b) = \prod_{i=1}^{L} p(y_i \mid c, b, y_{1:i-1}). \tag{1}$$

A one-pass block diffusion drafter does not expose these continuation-conditioned factors. Instead, it provides only per-position marginals $\{q_i\}_{i=1}^{L}$, where each $q_i(\cdot \mid c, b)$ predicts the token at position i from the shared conditioning context (c, b), without conditioning on the realized tokens at earlier positions within the same block. Accordingly, the natural distribution associated with a one-pass block diffusion drafter is the factorized distribution

$$Q(y_{1:L} \mid c, b) := \prod_{i=1}^{L} q_i(y_i \mid c, b). \tag{2}$$

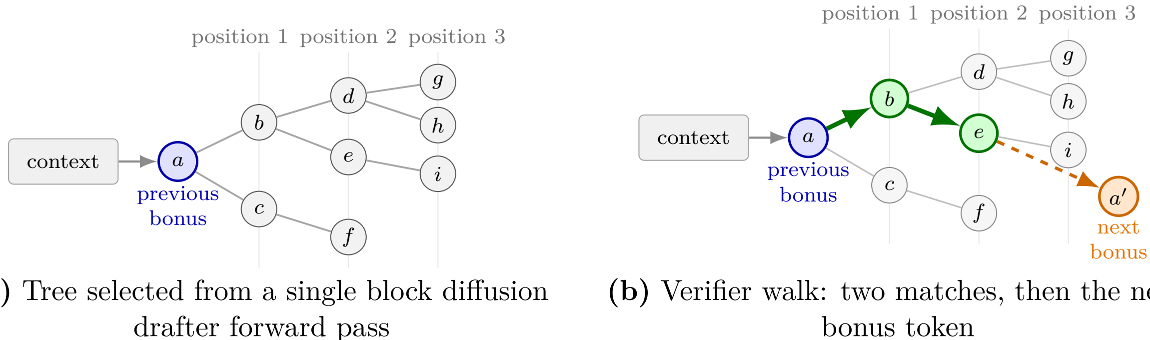
Thus, the target model provides a path-conditioned autoregressive distribution, whereas the drafter provides only a factorized distribution over the next L positions.

Our proposed DDTree builds on the factorized distribution (2): we use it to define the surrogate objective for selecting a draft tree from a single block diffusion drafter forward pass. As in standard speculative decoding with a block diffusion drafter, DDTree uses one lightweight drafter pass followed by target-model verification. The key difference is how we use the resulting one-pass marginals $\{q_i\}_{i=1}^{L}$. Rather than collapsing the marginals into a single continuation, we use them to construct a compact draft tree for verification.

4 The Proposed Method: Diffusion Draft Tree

4.1 Overview

DDTree builds a draft tree under a node budget B, in which a node at depth i represents a candidate token for the i-th future position. Ideally, in each decoding round, we would choose a draft tree with B nodes that maximizes the expected number of speculative tokens accepted by the target model. A single block diffusion drafter forward pass outputs, in parallel, one drafter-predicted marginal token distribution for each future position in the next block. It does not provide the target-conditioned continuation probabilities in (1) needed to optimize the expected target-model acceptance length directly. We therefore can only optimize a surrogate objective, which is the expected acceptance



(a) Tree selected from a single block diffusion drafter forward pass

(b) Verifier walk: two matches, then the next bonus token

Figure 2: Illustration of one DDTree decoding round. The bonus token a from the previous round is carried into the current round and serves as the tree root. One block diffusion drafter pass produces per-position marginals for the next three draft positions, from which DDTree selects a draft tree. During verification, the target model follows its own decoding rule, and the walk continues whenever the selected token appears as a child in the tree. In this example, two speculative nodes are matched (green), the matched path is added to the output, and the first unmatched target token a' becomes the bonus token for the next round.

length under the drafter's factorized approximation (2). In Section 4.2, we formalize this surrogate and derive the corresponding tree-construction objective.

At each decoding round, let c denote the full current context, including the prompt and all previously generated tokens. Recall that the bonus token b is already obtained by the target model: in the first round it comes from the prefill pass, and, in later rounds it is the previous round's bonus token. With this in place, conditioned on c and b, our proposed DDTree method implements speculative decoding for the next block by applying four steps:

1. Run the block diffusion drafter once to obtain per-position distributions for the next L draft positions after b.

2. Build a draft tree with B nodes from those distributions. The node budget B controls how many candidate tokens the verifier (target model) evaluates.

3. Compile the tree into input tensors for the target model and run one target-model forward pass with tree attention.

4. Walk the tree as illustrated in Figure 2: at each step, use the target model's decoding rule to choose the next token and check whether it matches a child in the tree; accept the matched drafted path, and carry the first unmatched target token (the next bonus token) to the next round.

4.2 Surrogate objective for draft-tree selection

Consider a decoding round with current context c and bonus token b. A single block diffusion drafter forward pass produces L per-position distributions $q_i(\cdot)$, $i = 1, \ldots, L$, for the next L positions after b. For $1 \le d \le L$, let

$$u = (u_1, \ldots, u_d) \in \mathcal{V}^d \tag{3}$$

denote a candidate continuation prefix after b. We identify each tree node with such a prefix, so a node at depth d represents the path $u_1, \ldots, u_d$ from the root b.[1]

A draft tree T is a set of such prefixes. The tree is valid if it is prefix-closed: whenever $(u_1, \ldots, u_d) \in T$ with $d \geq 2$, the parent prefix $(u_1, \ldots, u_{d-1})$ also lies in T. In other words, if a node is included in the tree, then so are all of its ancestors back to the root b. The root b itself does not count toward the node budget. Let

$$N = \sum_{d=1}^{L} |\mathcal{V}|^d$$

be the total number of nonempty prefixes of length at most L. In the regime relevant for DDTree, this number is enormous, and we always choose node budgets with $B \ll N$. Accordingly, throughout the rest of this section we treat $B < N$ as a standing assumption.

For a candidate continuation $y_{1:L}$, we define its acceptance length under T as the longest prefix of $y_{1:L}$ that appears in the tree:

$$\alpha_T(y_{1:L}) = \max\{d : y_{1:d} \in T\}, \tag{4}$$

with $\alpha_T(y_{1:L}) = 0$ when no depth-1 node matches. With this in place, the ideal tree-construction objective is

$$\max_{T:\,|T|\leq B,\, T \text{ valid}} \mathbb{E}_{Y_{1:L}\sim p(\cdot|c,b)}[\alpha_T(Y_{1:L})], \tag{5}$$

namely, the expected acceptance length under the target model p. This objective, however, is infeasible to optimize as it depends on the target-conditioned continuation probabilities (1), which are unavailable when the tree is constructed.

We therefore use a surrogate based on the factorized distribution induced by the block diffusion drafter $Q(\cdot \mid c, b)$, constructed solely from the per-position marginals $\{q_i\}_{i=1}^{L}$ in (2). This surrogate does not require full path-conditioned probabilities, which are unavailable from a single block diffusion forward pass.

Accordingly, we choose the draft tree by maximizing the expected acceptance length under the factorized distribution:

$$\max_{T:\,|T|\leq B,\, T \text{ valid}} \mathbb{E}_{Y_{1:L}\sim Q(\cdot|c,b)}[\alpha_T(Y_{1:L})]. \tag{6}$$

To analyze this surrogate objective, we express it in terms of prefix probabilities under Q. For a prefix $u = (u_1, \ldots, u_d)$, the probability that a sampled continuation under Q begins with u is

$$q(u \mid c, b) = \prod_{i=1}^{|u|} q_i(u_i \mid c, b). \tag{7}$$

The above is the key quantity for tree construction, because the acceptance length is determined by how far the sampled continuation matches a prefix in the tree. The next proposition shows that the surrogate objective decomposes as an additive sum of prefix masses over the nodes in T.

The above is the key quantity for tree construction, because the acceptance length is determined by the longest prefix of the sampled continuation that matches a path in the tree. The next proposition shows that the surrogate objective decomposes as an additive sum of prefix masses over the nodes in T.

[1] Throughout the paper, we use "node" to refer either to the full prefix u or, when the context is unambiguous, to its final token u_d.

Proposition 1. *For any valid draft tree T,*

$$\mathbb{E}_{Y_{1:L}\sim Q(\cdot|c,b)}[\alpha_T(Y_{1:L})] = \sum_{u\in T} q(u \mid c, b). \tag{8}$$

All proofs are in Appendix A. Proposition 1 shows that the surrogate objective is an additive sum of prefix probabilities $\sum_{u\in T} q(u \mid c, b)$. Therefore, selecting a draft tree reduces to choosing a valid set of at most B prefixes that maximizes this sum. Since the objective is additive over nodes and all terms are nonnegative, the optimal solution is to include the highest-probability prefixes up to the budget B, subject to the prefix-closure constraint. The next proposition shows that these top-B prefixes automatically form a valid tree and are therefore optimal.

For the next proposition and the results that follow, we assume $q_i(v \mid c, b) \in (0, 1)$ for all depths i and tokens v. This is the standard case when the draft model outputs probabilities via a softmax over finite logits. If needed, the construction can be modified slightly to handle edge cases without changing the main results.

Proposition 2. *Let $u^{(1)}, u^{(2)}, \ldots$ be all nonempty prefixes of length at most L, ordered so that*

$$q(u^{(1)} \mid c, b) \geq q(u^{(2)} \mid c, b) \geq \ldots,$$

with ties broken arbitrarily. Define

$$T_B = \{u^{(1)}, \ldots, u^{(B)}\}.$$

Then T_B is a valid draft tree with $|T_B| \leq B$. Moreover, T_B maximizes $\mathbb{E}_{Y_{1:L}\sim Q(\cdot|c,b)}[\alpha_T(Y_{1:L})]$ among all valid draft trees with $|T| \leq B$.

The above result is analogous to OPT-Tree's expected acceptance length objective [22], with the crucial difference that in our case all required probabilities are obtained from a single block diffusion drafter forward pass rather than from multiple autoregressive passes.

Remark 1. The result stated in Proposition 2 is exact for the surrogate objective induced by the factorized draft distribution $Q(\cdot \mid c, b)$, not for the true target-model distribution. The task here is not to recover the unavailable target model path-conditioned probabilities, but to make the best use of the information available from one block diffusion drafter forward pass.

4.3 Efficient and optimal tree construction

Proposition 2 reveals that an optimal draft tree is obtained by taking the B highest-probability prefixes. The remaining challenge is therefore algorithmic: how can we recover these prefixes efficiently, without enumerating the exponentially many possible prefixes up to depth L? In what follows, we show that this can be done with a simple best-first search procedure.

At each depth i, let $v_i^{(1)}, v_i^{(2)}, \ldots$ be the tokens ordered so that

$$q_i(v_i^{(1)} \mid c, b) \geq q_i(v_i^{(2)} \mid c, b) \geq \ldots,$$

and let $q_i^{(k)} = q_i(v_i^{(k)} \mid c, b)$. We index prefixes by token ranks rather than by vocabulary ids. Specifically, a rank tuple

$$\rho = (\rho_1, \ldots, \rho_d), \qquad 1 \leq d \leq L, \qquad 1 \leq \rho_i \leq |\mathcal{V}|,$$

denotes the depth-d prefix $(v_1^{(\rho_1)}, \dots, v_d^{(\rho_d)})$. For example, $\rho = (1, 3, 2)$ denotes the prefix obtained by taking the most probable token at position 1, the third-most probable token at position 2, and the second-most probable token at position 3. The probability of a prefix ρ is

$$q(\rho) = \prod_{i=1}^{d} q_i^{(\rho_i)},$$

and its log-probability is

$$\sigma(\rho) = \log q(\rho) = \sum_{i=1}^{d} \log q_i^{(\rho_i)}.$$

We will use $\sigma(\rho)$ to rank prefixes in a max-heap, which we use as a priority queue that returns the prefix with the largest value first. Taking logarithms converts products into sums, improving numerical stability while preserving the ordering of prefixes.

Let $K = \min(B, |\mathcal{V}|)$, and let

$$\mathcal{S}_K = \{(\rho_1, \dots, \rho_d) : 1 \le d \le L,\ 1 \le \rho_i \le K \text{ for } i = 1, \dots, d\}.$$

Thus, $\mathcal{S}_K$ contains exactly the prefixes that use only the top-K tokens at every depth.

Lemma 1. *There exists an optimal valid draft tree that maximizes the surrogate expected acceptance length in* (6)*, such that every node in the tree lies in $\mathcal{S}_K$. Equivalently, there exists an optimal valid draft tree in which every node uses only the top-K tokens at each depth.*

Lemma 1 reduces the search space to the top-K tokens at each depth. Algorithm 1 enumerates the retained prefixes $\mathcal{S}_K$ in descending order of $\sigma(\rho)$ using a max-heap. It starts from the tuple (1), i.e., the length-1 tuple containing only rank 1. When a tuple $\rho = (\rho_1, \dots, \rho_d)$ is popped, the algorithm generates at most two new tuples: its next sibling, which changes only the last rank to $\rho_d + 1$, and its first child, which appends rank 1 at the next depth. Intuitively, the sibling explores an alternative token at the current position, while the child extends the current prefix with the best available token at the next position. After B pops, the algorithm returns the top-B prefixes. The next proposition establishes the optimality of Algorithm 1.

Algorithm 1 Best-first draft-tree construction from one block diffusion drafter pass

Require: Top-K tokens $\{v_i^{(k)}\}_{i=1,k=1}^{L,K}$ and their probabilities $\{q_i^{(k)}\}_{i=1,k=1}^{L,K}$; node budget B
1: Initialize max-heap $H \leftarrow \{((1), \sigma((1)))\}$
2: Initialize draft tree $T \leftarrow \varnothing$
3: **while** $|T| < B$ and $H \neq \varnothing$ **do**
4: Pop the rank tuple $\rho = (\rho_1, \dots, \rho_d)$ with largest score $\sigma(\rho)$
5: Add prefix $(v_1^{(\rho_1)}, \dots, v_d^{(\rho_d)})$ to T
6: **if** $\rho_d + 1 \le K$ **then**
7: Push next sibling $(\rho_1, \dots, \rho_{d-1}, \rho_d + 1)$ with score $\sigma(\rho) - \log q_d^{(\rho_d)} + \log q_d^{(\rho_d+1)}$
8: **end if**
9: **if** $d < L$ **then**
10: Push first child $(\rho_1, \dots, \rho_d, 1)$ with score $\sigma(\rho) + \log q_{d+1}^{(1)}$
11: **end if**
12: **end while**
13: **return** draft tree T

Proposition 3. *Algorithm 1 returns an optimal valid draft tree for the surrogate objective in* (6) *under node budget B.*

Remark 2. The heap stage costs $O(B \log B)$: the algorithm performs at most B pops and at most $2B$ pushes, and the heap size is $O(B)$ throughout.

4.4 Efficient verification and cache update

Armed with an algorithm that efficiently constructs the optimal draft tree under the node budget, we now describe how the selected draft tree is compiled for verification, how the verifier walk is performed, and how the KV cache is updated afterward. To verify the selected draft tree in one target-model forward pass, we flatten it into a sequence of token ids rooted at the bonus token b. We assign position ids by tree depth so that the verifier applies the correct positional encoding. We then use tree attention [19], under which each drafted node attends to the past context through the KV cache and, within the drafted tree, only to the root, its ancestors, and itself. Together, these inputs let the verifier score all selected tree-branches in a single forward pass.

Verification then follows the target model's own decoding rule, whether greedy or temperature-based sampling. Figure 2(b) illustrates the process. Starting from the bonus token b, we check whether the token selected by the target model at the current node matches one of that node's children in the draft tree. If it does, that child is accepted and the walk continues. Since the verifier has already produced outputs for all drafted nodes in the same forward pass, this continuation requires no additional target-model call. If no child matches, the walk stops. The accepted path is appended to the output sequence, the first unmatched target-model token becomes the bonus token for the next round, and the KV cache is compacted to retain only the accepted path.

5 Experiments

We evaluate whether DDTree improves end-to-end speculative decoding over vanilla DFlash. Our experiments focus on decoding speed and acceptance behavior across target model size, domain, and decoding temperature.

5.1 Experimental setup

We evaluate three target models, Qwen3-4B, Qwen3-8B, and Qwen3-Coder-30B-A3B-Instruct [24], each paired with its corresponding DFlash checkpoint available at `https://huggingface.co/collections/z-lab/dflash`. Our benchmark suite spans reasoning tasks such as MATH-500 [17], GSM8K [8], AIME 2024, and AIME 2025; code tasks such as HumanEval [7], MBPP [3], LiveCodeBench [10], and SWE-bench Lite [12]; and general instruction and dialogue tasks such as MT-Bench [25] and Alpaca [21]. We run the benchmark on 8 H200 GPUs at temperatures 0.0 and 1.0 and report speedup relative to autoregressive decoding, mean acceptance length τ including the bonus token, and, for the case study below, the acceptance length histogram. Additional benchmark details, including per-dataset sample counts, decoding hyperparameters, warmup, and backend selection, are provided in Appendix B.

5.2 Main results

Figure 1 and Table 1 summarize the main benchmark results. Figure 1 summarizes the temperature 0.0 results across datasets and target models, using the best DDTree node budget for each dataset-model pair. Table 1 reports the exact numbers at both temperatures.

Table 1: Speedup over autoregressive decoding and mean acceptance length (τ). Results are reported separately for temperature 0.0 and temperature 1.0. For each dataset, model, and temperature, the DDTree entry uses the best node budget from $\{16, 32, 64, 128, 256, 512, 1024\}$.

Dataset	Qwen3-4B				Qwen3-8B				Qwen3-Coder-30B-A3B-Instruct			
	DFlash		DFlash+DDTree		DFlash		DFlash+DDTree		DFlash		DFlash+DDTree	
	Speedup	τ	Speedup	τ	Speedup	τ	Speedup	τ	Speedup	τ	Speedup	τ
Temperature = 0.0												
AIME 2024	5.56×	7.54	**7.27×**	**10.37**	5.38×	7.46	**7.35×**	**10.42**	3.95×	5.16	**5.93×**	**7.87**
AIME 2025	5.33×	7.37	**7.23×**	**10.23**	5.32×	7.39	**6.99×**	**9.86**	3.98×	5.13	**5.88×**	**7.63**
Alpaca	2.03×	3.11	**3.32×**	**5.35**	2.07×	3.12	**3.36×**	**5.09**	1.53×	2.22	**2.46×**	**3.55**
GSM8K	4.77×	6.51	**6.58×**	**9.33**	4.78×	6.57	**6.75×**	**9.54**	4.00×	5.18	**5.83×**	**7.55**
HumanEval	4.81×	6.62	**6.81×**	**9.44**	4.84×	6.61	**6.90×**	**9.67**	6.09×	8.02	**8.22×**	**10.72**
LiveCodeBench	4.97×	7.02	**6.78×**	**9.79**	5.02×	7.22	**7.10×**	**10.28**	4.72×	6.32	**6.54×**	**8.63**
MATH-500	5.54×	7.72	**7.50×**	**10.71**	5.56×	7.79	**7.52×**	**10.73**	4.29×	5.58	**6.21×**	**8.10**
MBPP	4.38×	6.10	**6.42×**	**9.16**	4.33×	5.99	**6.39×**	**9.07**	5.61×	7.19	**7.68×**	**9.94**
MT-Bench	2.64×	4.38	**4.18×**	**6.70**	2.56×	4.28	**4.10×**	**6.58**	2.04×	3.52	**3.27×**	**5.36**
SWE-bench Lite	2.70×	3.66	**4.25×**	**5.99**	2.65×	3.60	**4.23×**	**5.91**	2.77×	3.61	**4.38×**	**5.71**
Temperature = 1.0												
AIME 2024	3.50×	5.01	**5.31×**	**7.82**	3.46×	4.98	**5.36×**	**7.79**	3.12×	4.23	**4.94×**	**7.32**
AIME 2025	3.38×	4.79	**5.08×**	**7.22**	3.36×	4.79	**5.25×**	**7.71**	3.17×	4.26	**4.98×**	**6.62**
Alpaca	1.97×	2.96	**3.23×**	**4.97**	1.95×	2.94	**3.20×**	**4.89**	1.51×	2.14	**2.43×**	**3.51**
GSM8K	4.35×	5.99	**6.17×**	**8.87**	4.33×	5.93	**6.27×**	**8.95**	3.94×	5.08	**5.66×**	**7.38**
HumanEval	4.34×	5.97	**6.43×**	**9.18**	4.00×	5.48	**6.09×**	**8.59**	5.64×	7.60	**7.88×**	**10.42**
LiveCodeBench	4.53×	6.53	**6.44×**	**9.44**	4.52×	6.61	**6.46×**	**9.53**	3.77×	5.57	**5.47×**	**7.79**
MATH-500	4.65×	6.60	**6.60×**	**9.61**	4.56×	6.46	**6.59×**	**9.54**	4.01×	5.32	**5.91×**	**7.76**
MBPP	4.03×	5.56	**6.09×**	**8.72**	3.83×	5.30	**5.87×**	**8.34**	5.47×	7.03	**7.55×**	**9.76**
MT-Bench	2.46×	4.05	**3.91×**	**6.27**	2.30×	3.77	**3.75×**	**6.02**	1.96×	3.43	**3.10×**	**5.14**
SWE-bench Lite	2.29×	3.07	**3.71×**	**5.20**	2.10×	2.82	**3.47×**	**4.86**	2.42×	3.16	**3.80×**	**4.96**

DDTree improves every entry in Table 1, covering all $10 \times 3 \times 2 = 60$ dataset-model-temperature settings. The gains are consistent across all three target models, across reasoning, code, and general instruction tasks, and at both temperatures.

5.3 Budget-quality tradeoff

Figure 3 shows a case study on MATH-500 with Qwen3-8B at temperature 0.0. As our DDTree node budget grows, acceptance length increases steadily, and the end-to-end speedup improves until it peaks around budgets of 256 to 512. Pushing the budget to 1024 increases acceptance length further, but the tradeoff is no longer favorable as the additional overhead of verifying more drafted tokens outweighs the gain from the longer accepted prefix. Vanilla DFlash uses a single block of size 16, so the figure also shows that DDTree provides better speedup under the same conceptual budget. This highlights the importance of a front-heavy tree that does not waste budget on low-probability trajectories. Note that the optimal budget can shift across hardware platforms and implementations.

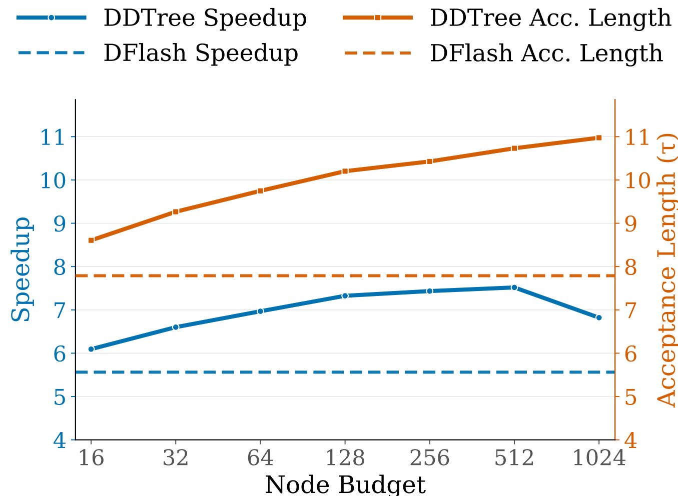


Figure 3: Budget tradeoff on MATH-500 with Qwen3-8B at temperature 0.0. Acceptance length increases steadily with the DDTree node budget, while speedup peaks at an intermediate budget once verifier cost becomes dominant.

5.4 Acceptance length distribution

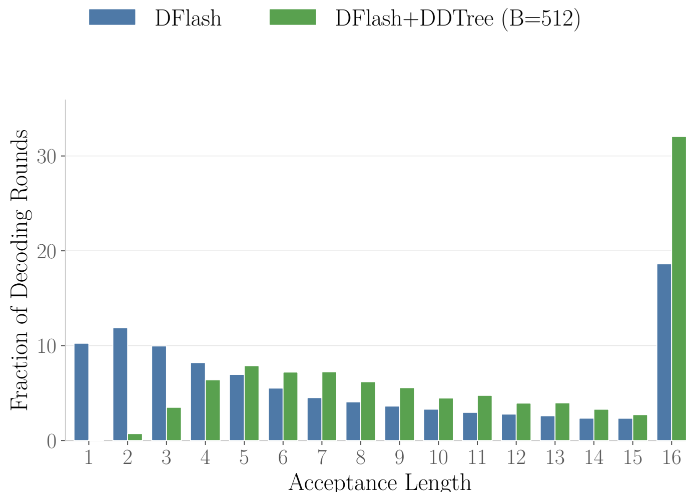


Figure 4: Acceptance length distribution on MATH-500 with Qwen3-8B at temperature 0.0. The DDTree histogram uses the best speedup budget, $B = 512$, and shifts mass toward longer accepted prefixes, especially full block acceptances.

Figure 4 shows the histogram of acceptance lengths on MATH-500 with Qwen3-8B at temperature 0.0. The plotted DDTree distribution uses the best speedup budget, $B = 512$. Compared with vanilla DFlash, DDTree shifts substantial probability mass toward longer accepted prefixes. With DDTree, it becomes much rarer to observe acceptance lengths below 4, while full-block acceptance

at length 16 becomes substantially more common. This shift explains the end-to-end speedup improvement: DDTree makes long accepted prefixes substantially more common, so the verifier needs fewer rounds per generated token.

Acknowledgments

L. R. and Y. R. were supported by the European Union (ERC, SafetyBounds, 101163414). Views and opinions expressed are however those of the authors only and do not necessarily reflect those of the European Union or the European Research Council Executive Agency. Neither the European Union nor the granting authority can be held responsible for them. This research was also partially supported by the Israel Science Foundation (ISF grant 729/21). Y. R. acknowledges additional support from the Career Advancement Fellowship at the Technion. The contribution of the first author is part of a PhD thesis research conducted at the Technion.

References

[1] Zihao An, Huajun Bai, Ziqiong Liu, Dong Li, and Emad Barsoum. PARD: Accelerating LLM inference with low-cost PARallel draft model adaptation. In *The Fourteenth International Conference on Learning Representations*, 2026.

[2] Marianne Arriola, Subham Sekhar Sahoo, Aaron Gokaslan, Zhihan Yang, Zhixuan Qi, Jiaqi Han, Justin T Chiu, and Volodymyr Kuleshov. Block diffusion: Interpolating between autoregressive and diffusion language models. In *The Thirteenth International Conference on Learning Representations*, 2025.

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

[4] Tianle Cai, Yuhong Li, Zhengyang Geng, Hongwu Peng, Jason D. Lee, Deming Chen, and Tri Dao. Medusa: Simple LLM inference acceleration framework with multiple decoding heads. In *Forty-first International Conference on Machine Learning*, 2024.

[5] Charlie Chen, Sebastian Borgeaud, Geoffrey Irving, Jean-Baptiste Lespiau, Laurent Sifre, and John Jumper. Accelerating large language model decoding with speculative sampling. *arXiv preprint arXiv:2302.01318*, 2023.

[6] Jian Chen, Yesheng Liang, and Zhijian Liu. Dflash: Block diffusion for flash speculative decoding. *arXiv preprint arXiv:2602.06036*, 2026.

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

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

[9] Tri Dao. Flashattention-2: Faster attention with better parallelism and work partitioning. In *The Twelfth International Conference on Learning Representations*, 2024.

[10] 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. In *The Thirteenth International Conference on Learning Representations*, 2025.

[11] Wonseok Jeon, Mukul Gagrani, Raghavv Goel, Junyoung Park, Mingu Lee, and Christopher Lott. Recursive speculative decoding: Accelerating LLM inference via sampling without replacement. In *ICLR 2024 Workshop on Large Language Model (LLM) Agents*, 2024.

[12] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*, 2024.

[13] Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding. In *International Conference on Machine Learning*, pages 19274–19286. PMLR, 2023.

[14] Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. EAGLE: Speculative sampling requires rethinking feature uncertainty. In *Forty-first International Conference on Machine Learning*, 2024.

[15] Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. Eagle-2: Faster inference of language models with dynamic draft trees. In *Proceedings of the 2024 conference on empirical methods in natural language processing*, pages 7421–7432, 2024.

[16] Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. EAGLE-3: Scaling up inference acceleration of large language models via training-time test. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025.

[17] Hunter Lightman, Vineet Kosaraju, Yuri Burda, Harrison Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step. In *The twelfth international conference on learning representations*, 2023.

[18] Fuliang Liu, Xue Li, Ketai Zhao, Yinxi Gao, Ziyan Zhou, Zhonghui Zhang, Zhibin Wang, Wanchun Dou, Sheng Zhong, and Chen Tian. Dart: Diffusion-inspired speculative decoding for fast llm inference. *arXiv preprint arXiv:2601.19278*, 2026.

[19] Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Zeyu Wang, Zhengxin Zhang, Rae Ying Yee Wong, Alan Zhu, Lijie Yang, Xiaoxiang Shi, et al. Specinfer: Accelerating generative large language model serving with tree-based speculative inference and verification. *arXiv preprint arXiv:2305.09781*, 2023.

[20] Benjamin Frederick Spector and Christopher Re. Accelerating LLM inference with staged speculative decoding. In *Workshop on Efficient Systems for Foundation Models @ ICML2023*, 2023.

[21] Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B Hashimoto. Alpaca: A strong, replicable instruction-following model. *Stanford Center for Research on Foundation Models. https://crfm. stanford. edu/2023/03/13/alpaca. html*, 3(6):7, 2023.

[22] Jikai Wang, Yi Su, Juntao Li, Qingrong Xia, Zi Ye, Xinyu Duan, Zhefeng Wang, and Min Zhang. Opt-tree: Speculative decoding with adaptive draft tree structure. *Transactions of the Association for Computational Linguistics*, 13:188–199, 2025.

[23] Yunfan Xiong, Ruoyu Zhang, Yanzeng Li, and Lei Zou. Dyspec: Faster speculative decoding with dynamic token tree structure. *World Wide Web*, 28(3):36, 2025.

[24] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

[25] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in neural information processing systems*, 36:46595–46623, 2023.

A Mathematical proofs

Proof of Proposition 1. We expand

$$\alpha_T(Y_{1:L}) = \sum_{d=1}^{L} \mathbf{1}\{\alpha_T(Y_{1:L}) \geq d\}. \tag{9}$$

For a fixed depth d, the event $\{\alpha_T(Y_{1:L}) \geq d\}$ holds if and only if the sampled depth-d prefix $Y_{1:d}$ is one of the depth-d nodes in T. These depth-d events are disjoint because $Y_{1:d}$ can equal only one sequence. Therefore

$$\Pr[\alpha_T(Y_{1:L}) \geq d] = \sum_{u \in T: |u|=d} \Pr[Y_{1:d} = u] = \sum_{u \in T: |u|=d} q(u \mid c, b). \tag{10}$$

Summing over d gives the claim. □

Proof of Proposition 2. Let v be any strict descendant of u. Then

$$q(v \mid c, b) = q(u \mid c, b) \prod_{i=|u|+1}^{|v|} q_i(v_i \mid c, b) < q(u \mid c, b). \tag{11}$$

Therefore, every strict ancestor has a strictly larger probability than any of its descendants. It follows that, in any nonincreasing ordering of prefixes, every ancestor must appear before its descendants, which implies that T_B is prefix-closed and hence valid.

For optimality, Proposition 1 shows that the surrogate objective for a tree T is

$$\sum_{u \in T} q(u \mid c, b).$$

This objective is additive over nodes, and every term is nonnegative. Therefore, among all sets of at most B prefixes, the maximum is attained by taking the top-B probabilities $q(u \mid c, b)$, namely the prefixes in T_B. Since T_B is valid, it is optimal among all valid draft trees with $|T| \leq B$. If ties occur between unrelated prefixes, other optimal trees may also exist. □

Proof of Lemma 1. Order all nonempty prefixes by nonincreasing $q(u \mid c, b)$, breaking ties by placing prefixes in $\mathcal{S}_K$ before prefixes outside $\mathcal{S}_K$. By Proposition 2, the first B prefixes in this order form an optimal valid draft tree. It therefore suffices to show that all of these first B prefixes lie in $\mathcal{S}_K$.

If $B \geq |\mathcal{V}|$, then $K = \min(B, |\mathcal{V}|) = |\mathcal{V}|$, so every prefix already lies in $\mathcal{S}_K$. Thus only the case $B < |\mathcal{V}|$ remains, in which case $K = B$.

Assume by contradiction that some prefix $u = (u_1, \ldots, u_d) \notin \mathcal{S}_K$ appears among the first B prefixes. Let

$$I = \{j \in \{1, \ldots, d\} : \text{the rank of } u_j \text{ at depth } j \text{ exceeds } B\},$$

pick any $i \in I$, and let $\tilde{u}$ be obtained from u by replacing u_j with $v_j^{(1)}$ for every $j \in I$. For each $k \in \{1, \ldots, B\}$, let $u^{(k)}$ be obtained from $\tilde{u}$ by replacing its i-th coordinate with $v_i^{(k)}$.

Then $u^{(1)}, \ldots, u^{(B)}$ are B distinct prefixes in $\mathcal{S}_K$, and for every k,

$$q(u^{(k)} \mid c, b) \geq q(u \mid c, b),$$

since each replaced coordinate is changed to a token of rank at most B. By the tie-breaking rule, each $u^{(k)}$ appears before u in the ordering. Hence at least B prefixes appear before u, contradicting that u is among the first B.

Therefore no prefix outside $\mathcal{S}_K$ can appear among the first B prefixes. Hence all first B prefixes lie in $\mathcal{S}_K$, and so some optimal valid draft tree is contained in $\mathcal{S}_K$. □

Proof of Proposition 3. Each $\rho \in \mathcal{S}_K$ corresponds to the prefix $(v_1^{(\rho_1)}, \dots, v_d^{(\rho_d)})$, and its score is

$$\log q(\rho) = \sum_{i=1}^{d} \log q_i^{(\rho_i)}.$$

By Lemma 1, some optimal valid draft tree is contained in $\mathcal{S}_K$. It is therefore enough to show that Algorithm 1 returns the B highest-scoring elements of $\mathcal{S}_K$, in nonincreasing order of $q(\rho)$.

For every $\rho = (\rho_1, \dots, \rho_d) \in \mathcal{S}_K$ other than (1), define its predecessor by

$$\text{pred}(\rho) = \begin{cases} (\rho_1, \dots, \rho_{d-1}, \rho_d - 1), & \rho_d > 1, \\ (\rho_1, \dots, \rho_{d-1}), & \rho_d = 1, \ d > 1. \end{cases}$$

This predecessor is unique. If $\rho_d > 1$, then popping $\text{pred}(\rho)$ generates ρ as the next sibling. If $\rho_d = 1$ and $d > 1$, then popping $\text{pred}(\rho)$ generates ρ as the first child.

We next show that the heap pops tuples in nonincreasing order of $q(\rho)$. The first pop is (1), since it is the unique initial heap element. Now consider any later iteration. At that point, the heap contains exactly the unpopped tuples whose predecessor has already been popped. Fix any unpopped tuple ρ, and let $\bar{\rho}$ be the first tuple on the predecessor chain from ρ back toward (1) whose predecessor has already been popped. Then $\bar{\rho}$ is in the heap. Moreover,

$$q(\text{pred}(\rho)) \geq q(\rho)$$

for every noninitial tuple ρ: in the sibling case this replaces $q_d^{(\rho_d)}$ by the larger quantity $q_d^{(\rho_d - 1)}$, and in the child case it removes the factor $q_d^{(1)} < 1$. Applying this inequality repeatedly along the chain from ρ to $\bar{\rho}$ gives $q(\bar{\rho}) \geq q(\rho)$. Therefore the maximum-probability unpopped tuple is always present in the heap, and each pop removes a highest-probability remaining tuple. Repeating this argument proves that the pop sequence is nonincreasing in $q(\rho)$.

After B pops, the returned set T therefore consists of the top-B prefixes. It remains to show that T is valid. Let $\rho = (\rho_1, \dots, \rho_d) \in T$ with $d \geq 2$. Repeatedly applying pred to ρ decreases the last coordinate until reaching $(\rho_1, \dots, \rho_{d-1}, 1)$, and one more predecessor step yields the parent tuple $(\rho_1, \dots, \rho_{d-1})$. Since each tuple is generated only after its predecessor is popped, every tuple on this chain was popped earlier and therefore belongs to T. Hence the parent of every returned nonroot prefix is also returned, so T is prefix-closed.

After B pops, the returned set T is exactly the top-B elements of $\mathcal{S}_K$. By Proposition 2, T is a valid draft tree and is optimal among trees contained in $\mathcal{S}_K$. By Lemma 1, some global optimum is contained in $\mathcal{S}_K$. Hence T is optimal for (6).

Finally, Proposition 1 implies that, within $\mathcal{S}_K$, the surrogate objective is maximized by taking the highest-probability prefixes. Since Algorithm 1 returns exactly those prefixes, and some global optimum lies in $\mathcal{S}_K$, the returned tree is optimal for (6) under node budget B. □

B Benchmark details

All runs use block size 16, DDTree node budgets $\{16, 32, 64, 128, 256, 512, 1024\}$, temperatures 0.0 and 1.0, a maximum of 2048 new tokens, and bfloat16 inference. We run the benchmark on 8 H200 GPUs and shard the evaluation set across workers. Before timing the benchmark loop, we run a short warmup prompt through the autoregressive baseline, vanilla DFlash, and each DDTree budget so that one time setup costs are excluded from the reported measurements.

For DDTree, the target model uses standard PyTorch scaled dot product attention, because FlashAttention-2 [9] does not support the required tree attention pattern. The DFlash drafter itself still uses FlashAttention-2. For fairness, for the autoregressive baseline and vanilla DFlash, we evaluate the target model with both standard PyTorch scaled dot product attention and FlashAttention-2, and report the faster result, which can only improve these baselines relative to DDTree.

Table 2 lists the number of evaluated examples for each dataset. We follow the original DFlash benchmark setup for these sample counts.

Table 2: Number of evaluated examples per dataset in the benchmark suite.

Dataset	**Examples**
AIME 2024	30
AIME 2025	30
Alpaca	128
GSM8K	128
HumanEval	164
LiveCodeBench	128
MATH-500	128
MBPP	128
MT-Bench	80
SWE-bench Lite	128