SAM Decoding: Speculative Decoding via Suffix Automaton

Yuxuan Hu[1,2], **Ke Wang**[1,2], **Xiaokang Zhang**[1,2], **Fanjin Zhang**[4]
Cuiping Li[1,3], **Hong Chen**[1,3], **Jing Zhang**[1,3]*
[1]School of Information, Renmin University of China, Beijing, China
[2]Key Laboratory of Data Engineering and Knowledge Engineering, Beijing, China
[3]Engineering Research Center of Database and Business Intelligence, Beijing, China
[4]Knowledge Engineering Group, Tsinghua University, Beijing, China

Abstract

Speculative decoding (SD) has been demonstrated as an effective technique for lossless LLM inference acceleration. Retrieval-based SD methods, one kind of model-free method, have yielded promising speedup, but they often rely on single retrieval resources, inefficient retrieval methods, and are constrained to certain tasks. This paper presents a novel retrieval-based speculative decoding method that adapts suffix automaton (SAM) for efficient and accurate draft generation by utilizing the generating text sequence and static text corpus. Unlike existing n-gram matching methods, SAM-Decoding finds the exact longest suffix match, achieving an average time complexity of O(1) per generation step of SAM update and suffix retrieval. It can also integrate with existing methods, adaptively selecting a draft generation strategy based on match length to generalize to broader domains. Extensive experiments on Spec-Bench show that our method is 18%+ faster than other retrieval-based SD methods. Additionally, when combined with advanced EAGLE-2, it provides an additional speedup of 3.28% – 11.13% across various-sized LLM backbones. Our code is available at our repository.

1 Introduction

The Transformer-based Large Language Models (LLMs) (Brown et al., 2020; Dubey et al., 2024; Yang et al., 2024) have demonstrated remarkable abilities and are extensively adopted in numerous domains. The scaling law drives LLMs to become deeper, reaching hundreds of billions of parameters, which makes them inefficient for generating text in a token-by-token autoregressive manner. Speculative decoding (SD) methods (Leviathan et al., 2023; Cai et al., 2024) seek to tackle this problem by quickly generating multiple draft tokens and subsequently concurrently verifying them with LLMs.

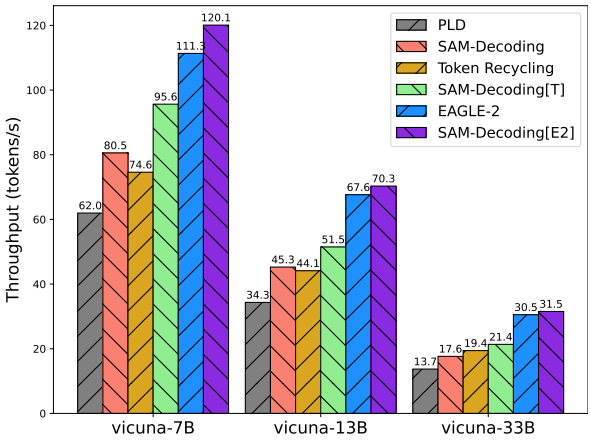

Figure 1: Throughput of Vicuna-7B, Vicuna-13B, Vicuna-33B on MT-Bench with A6000 GPU using PLD, Token Recycling, EAGLE-2, and SAM-Decoding variant, where SAM-Decoding is our proposed method.

These methods can decrease inference latency substantially while maintaining decoding accuracy.

Most speculative methods can be categorized into generation-based methods. For these methods, one or more small-sized draft models need to be carefully chosen and trained. For example, Medusa (Cai et al., 2024) utilizes multiple decoding heads to generate multiple future tokens while EAGLE-2 (Li et al., 2024b) leverages shallow Transformer layers to predict the next last hidden states and corresponding decoding tokens. Token cycling (Luo et al., 2024) is a special case of generation-based methods that dynamically maintain the posterior distribution of each token, resulting in a model-free generative approach. Although these methods achieve impressive speedup, they often fail to generate long draft tokens due to drafting overhead or decaying prediction accuracy. Retrieval-based speculative decoding methods, a major type of model-free method, aim to remedy this issue by generating draft tokens from the existing text corpus or the current text sequence.

However, current retrieval-based methods have

*Corresponding author. zhang-jing@ruc.edu.cn

Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 12187–12204
July 27 - August 1, 2025

notable limitations. **Firstly**, diverse retrieval sources contribute to the efficiency of retrieval-based SD methods, but existing methods typically rely on a single retrieval source: PLD (Saxena, 2023) focuses on current text while REST (He et al., 2024) uses a static text corpus. **Secondly**, restricted by n-gram matching, the retrieval methods used in existing methods have limitations in efficiency and accuracy. As an example, PLD finds n-gram matching from the current text sequence via brute force, it has poor theoretical computational complexity and limited applicability to larger text corpus. **Thirdly**, the limited integration between retrieval-based and generation-based methods restricts their inference speed. For instance, PLD is capable of producing highly effective drafts in a few positions but performs poorly in others. On the other hand, generation-based methods, such as Token Recycling and EAGLE-2, are able to generate quality drafts across most positions. Consequently, integrating retrieval-based methods with generation-based approaches has the potential to further enhance their inference speed.

To address limitations in previous retrieval-based methods, this paper introduces SAM-Decoding, an innovative speculative decoding technique based on suffix automaton. (1) To enhance the coverage of the retrieved corpus, we utilize both the generating text sequence and the static text corpus as retrieved sources. (2) To improve the retrieval efficiency and accuracy, we adapt a suffix automaton (SAM) to solve the longest suffix match problem, which yields more accurate match positions and exact match lengths compared to n-gram matching. As for retrieval efficiency, the average time complexity of SAM update and suffix retrieval is $O(1)$ by capturing relationships between adjacent suffixes. (3) To combine the retrieval-based method and the generation-based method, we adaptively select either the retrieval method or the generation method to provide drafts at each position based on the match length of the automaton, which can better utilize the advantages of retrieval-based methods and generation-based methods.

Specifically, SAM-Decoding creates both a dynamic suffix automaton for the generating text sequence and a static suffix automaton for the text corpus. The nodes of the suffix automaton represent substrings in the text sequence and text corpus. The earliest position of each substring is recorded in each node. During generation, we can directly retrieve drafts from the automaton using the match position and the match length. After each generation step, for the static automaton, the match position is updated, while for the dynamic automaton, its structure and the match position are updated simultaneously. The primary contributions of this work are as follows.

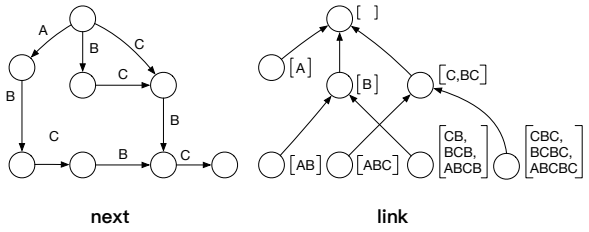

Figure 2: The suffix automaton corresponding to the string "ABCBC".

- We introduce a model-free, retrieval-based SD method leveraging the suffix automaton, which incorporates multiple retrieval sources and achieves higher efficiency compared to existing approaches.
- We propose integrating retrieval-based methods with generation-based methods by utilizing the match length of the automaton from retrieval methods, enabling better exploitation of the strengths of both approaches.
- Extensive evaluations demonstrate the competitive performance of our method across tasks. On Spec-Bench, SAM-Decoding achieves $18\%+$ faster than previous retrieval-based speculative decoding methods (e.g., PLD, REST, etc.). When combined with EAGLE-2 (Li et al., 2024b), as shown in Figure 1, our method outperforms the state-of-the-art, delivering an additional 3.28% – 11.13% speedup on MT-Bench w.r.t. various LLM backbones.

2 Background

2.1 Suffix Automaton

Suffix Automaton is an efficient data structure for representing the substring index of a given string, which allows fast substring retrieval. The time complexity of constructing a suffix automaton is $O(L)$, where L is the length of the string and it can be constructed incrementally.

As shown in Figure 2, a suffix automaton contains a series of nodes and two types of edges, **extension edges (next)** and **suffix link edges (link)**.

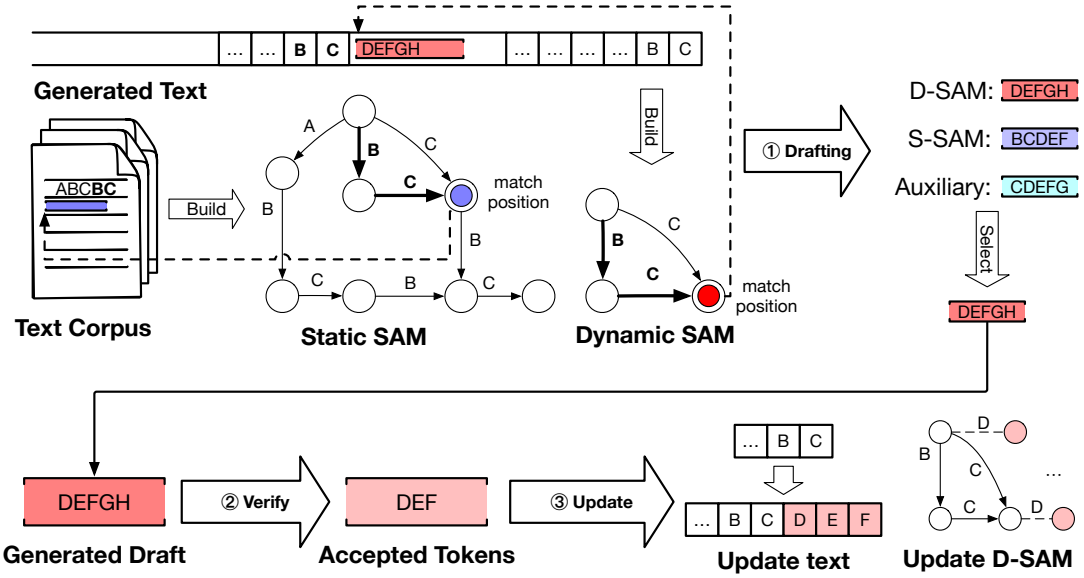

Figure 3: Overview of SAM-Decoding's workflow. In each round of generation, the suffix automaton matches the suffixes of the generating text and retrieves the draft from the text corpus and the generated text respectively according to the matching position. Our method can be combined with an auxiliary SD algorithm (Auxiliary) to deal with the scenarios where the retrieval is not applicable. We select the best draft from the three candidate drafts based on the match length, and then the drafts are verified by the LLM for accepted tokens. Using these accepted tokens, we finally extend the dynamic SAM and generate text for the next round of generation.

A node in the automaton corresponds to all substrings that have the same ending position in the string. Meanwhile, extension edges are standard edges that represent a possible extension of the current substring by appending a new token. In contrast, suffix link edges create a path that allows the automaton to quickly jump to states representing shorter suffixes of the current substring.

Based on the two types of transfer edges, for a progressively generated token sequence, we can find the longest suffix that matches the sequence in a suffix automaton at each generation step with an average $O(1)$ time complexity. To better understand the matching process, it can be viewed as starting from the root node and then transitioning based on tokens generated by the LLM. The final node reached represents the matching result.

2.2 Speculative Decoding

Given the model input $x = (x_1, x_2, \ldots, x_t)$, an LLM generates a new token x_{t+1} at each generation step autoregressively. The key idea of speculative decoding is to utilize a lightweight draft model to generate multiple candidate tokens quickly, i.e., $x_{\text{draft}} = (x_{t+1}, x_{t+2}, \ldots, x_{t+n})$, and then the target LLM simultaneously evaluates these candidates and accept those aligned with the output distribution of the LLM, i.e., $x_{\text{accept}} = (x_{t+1}, x_{t+2}, \ldots, x_{t+m})$, where n and m denote the size of the draft and the number of accepted tokens.

In the above, we assume that the draft is a sequence of tokens. Recent works proposed to verify a candidate token tree via a tree mask in the attention module to make the target LLM simultaneously evaluate multiple branches of this token tree, thereby increasing the acceptance length of the draft model.

3 SAM-Decoding

In this section, we introduce our proposed method, SAM-Decoding. SAM-Decoding is a retrieval-based speculative decoding method designed to address three key limitations in existing retrieval-based speculative methods: **(1)** The use of insufficient retrieval sources. **(2)** The employment of inefficient retrieval methods and restrictions on n-gram matching lengths. **(3)** Lack of integration with generation-based methods

To tackle the first two limitations, SAM-Decoding leverages suffix automaton on diverse text sources, which significantly enhances the coverage of retrieved corpus and the efficiency of the retrieval process while allowing for flexible matching lengths (Section 3.1, 3.2, and 3.3). In what follows, we detail how SAM-Decoding can be integrated with generation-based methods (Sec-

tion 3.4). By utilizing the precise matching information provided by the suffix automaton, our method not only overcomes the third limitation but also ensures consistent performance improvements across a wide range of tasks. The workflow of SAM-Decoding is shown in Figure 3.

3.1 Suffix Automaton Construction

To cover comprehensive retrieval sources, SAM-Decoding builds suffix automaton (SAM) by utilizing the generating text sequence and static text corpus. Thus, we construct two types of suffix automatons: a **dynamic suffix automaton** and a **static suffix automaton**.

For the generating text sequence, we create and expand a dynamic suffix automaton incrementally as generation progresses and perform text matching concurrently. At each node of the dynamic suffix automaton, we record the earliest position of all substrings corresponding to that node in the reference string, termed as **min_endpos**, which allows us to efficiently locate the previous ending position of the matched longest suffix. Hereafter, the subsequent tokens after the matched suffix can be regarded as potential drafts.

For the static text corpus, we pre-build a static suffix automaton offline, which is used for text matching during inference. At each node of the static suffix automaton, we compute the top-k successor tokens of each node, termed as **topk_succ**, and subsequently use them to construct tree-structured drafts. Although computing the successor token requires additional computation, this can be done offline, eliminating the need to account for this time overhead in real-time processing.

A suffix automaton can be constructed in linear time using Blumer's algorithm (Blumer et al., 1984). Since the suffix automaton is designed for a single text, to this end, for static suffix automation, we concatenate multiple strings in the corpus by using special symbols like an End-of-Sentence (EOS) token and then construct a static suffix automaton for this concatenated string. The construction process of the suffix automaton is detailed in Appendix A.1.

3.2 Drafting with Suffix Automaton

Let S denote the suffix automaton, T denote its associated reference text, and $x = (x_1, x_2, \ldots, x_t)$ denote the current text sequence. The state within the suffix automaton corresponding to the sequence x is denoted as s_t (including match position and match length). The initial state s_0 corresponds to the root node of the suffix automaton. As shown in Algorithm 1, in each round of generation, the transition to the next state is performed based on the newly generated token x_{t+1} and the current state s_t:

$$s_{t+1} = \text{Transfer}(S, x_{t+1}, s_t).$$

Then, for dynamic suffix automaton, we extract n consecutive tokens from the reference text T to form a draft, using the **min_endpos** value stored in the node corresponding to state s_{t+1}, termed as p_{t+1}. Then the draft d_{t+1} is defined as:

$$d_{t+1} = T[p_{t+1} : p_{t+1} + n],$$

where d_{t+1} represents the generated draft and n denotes the length of the draft.

And for static suffix automaton, we construct a tree-structured draft by Prim's algorithm based on top-k successors, as detailed in Appendix A.4,

$$d_{t+1} = \text{Prim}(S, s_{t+1}, x_{t+1}).$$

We use different draft generation strategies for dynamic suffix automaton and static suffix automaton, this is due to the dynamic automaton preferring to generate high quality drafts at few positions, while the static automaton prefers to generate average-quality drafts at most positions, so we use sequence-structured drafts for the dynamic automaton to enhance max accept length, while uses tree-structured drafts for the static automaton to enhance average accept length.

We track the match length (denoted as l) to determine whether to use the static suffix automaton or the dynamic suffix automaton. Specifically, let l_1 and l_2 be the matching lengths of the static and dynamic automata, respectively. We use the draft from the static automaton if $l_1 > l_2 + l_{\text{bias}}$, otherwise, we use the draft from the dynamic automaton, where l_{bias} is a predefined constant.

3.3 Update of Suffix Automaton

After the draft is generated, we verify it using the LLM and accept the correct tokens, denoted as $x_{\text{accept}} = (x_{t+1}, x_{t+2}, \ldots, x_{t+m})$. We then update the match state and suffix automaton based on these accepted tokens. For the dynamic suffix automaton, we transfer the match state and update the automaton simultaneously. Let S_t denote the

Algorithm 1 State Transfer of Suffix Automaton

function Transfer
  Input: suffix automaton S, next token t, current state s, current matching length l
  while $s \neq S$.root **and** $t \notin s$.next **do**
    $s,\ l = s$.link, s.link.length
  end while
  if $t \in s$.next **then**
    $s,\ l = s$.next$[t],\ l + 1$
  else
    $l = 0$
  end if
  Output: next state s, next match length l
end function

dynamic suffix automaton for the generated text $(x_1, x_2, \ldots, x_t)$. The process is as follows:

$$s_{t+i} = \text{Transfer}(S_{t+i-1}, s_{t+i-1}, x_{t+i}),$$
$$S_{t+i} = \text{Expand}(S_{t+i-1}, x_{t+i}),$$
$$i \in \{1, 2, ..., m\},$$

For the static suffix automaton, we transfer to new states without updating the automation:

$$s_{t+i} = \text{Transfer}(S, s_{t+i-1}, x_{t+i}),\ i \in \{1, 2, ..., m\}.$$

The process of transfer is detailed in Algorithm 1 and the process of updating the suffix automaton is detailed in Appendix A.1.

Using amortized analysis, we can prove that the average complexity of state transfer is $O(1)$, where L is the length of the current generated text (C.f. proof in Appendix A.5). Existing methods like PLD uses a brute-force search for n-gram matches, resulting in a time complexity of $O(n^2 L)$. REST also employs n-grams but searches using suffix arrays, leading to a time complexity of $O(n^2 \log C)$. Here, n is the predefined maximum matching length, L is the length of the current text, and C is the size of the static corpus. Therefore, our proposed SAM-Decoding has a lower time complexity and can find the exact longest suffix match without any limit on matching length, making it faster and more accurate for draft generation.

3.4 Generation-based Method Integration

The retrieval-based speculative decoding methods excel at generating drafts from the corpus or the current text sequence effectively. If it fails to produce a satisfactory draft, other speculative decoding techniques can be employed to generate more diverse drafts. To combine different types of drafts, a straightforward idea is that the length of the suffix match can indicate the confidence of the draft produced by the automaton, where long matches imply that more tokens are likely to be acceptable.

To implement this, we concurrently use an auxiliary speculative decoding technique alongside the suffix automaton. During each generation step, we adaptively select the drafts offered by the automaton or the auxiliary SD method based on the match length of the generated text within the automaton. For the auxiliary SD method, we set a fixed virtual match length $l_{\text{threshold}}$. In our study, we consider two auxiliary cutting-edge speculative decoding methods: the model-free Token Recycling and the model-based EAGLE-2. Among them, Token Recycling maintains an adjacency list of the top-k probable next tokens for each token. It builds a draft tree using breadth-first search and continuously updates the list based on the latest tokens. EAGLE-2, on the other hand, leverages a Transformer decoder layer to jointly predict the last hidden states of the LLM and the next token autoregressively.

4 Experiments

In this section, we first introduce our experimental setup, then present the experimental results, and finally present the ablation experiments.

Models and Tasks. We conducted experiments on Vicuna-7B-v1.3 (Zheng et al., 2023). We evaluated SAM-Decoding on Spec-Bench (Xia et al., 2024), HumanEval (Chen et al., 2021), and HARGID (Kamalloo et al., 2023). Spec-Bench is a comprehensive benchmark designed for assessing SD methods across diverse scenarios. It is based on six commonly used datasets, MT-Bench (Zheng et al., 2023), WMT14 DE-EN, CNN/Daily Mail (Nallapati et al., 2016), Natural Question (Kwiatkowski et al., 2019), GSM8K (Cobbe et al., 2021), and DPR (Karpukhin et al., 2020), including six aspects: Multi-turn Conversation (MT), Translation (Trans), Summarization (Sum), Question Answering (QA), Mathematical Reasoning (Math), and Retrieval-augmented Generation (RAG). In addition, HumanEval, and HARGID are used to evaluate the speed of SD methods in the Code Generation task and Context Q&A task, respectively.

Baselines. We considered the following baseline methods, including the model-based method

Method	Spec-Bench			HumanEval			HAGRID		
	#MAT	Tokens/s	Speedup	#MAT	Tokens/s	Speedup	#MAT	Tokens/s	Speedup
Lookahead	1.63	44.37	1.20×	1.76	30.81	1.54×	1.46	23.58	1.32×
REST	1.63	51.34	1.38×	1.85	34.60	1.74×	1.53	24.91	1.39×
PIA	2.08	55.45	1.47×	2.62	65.49	1.68×	2.43	66.65	1.95×
PLD	1.75	59.02	1.56×	1.65	59.04	1.52×	2.03	44.11	1.29×
SAM-Decoding	**2.30**	**69.37**	**1.84**×	**2.64**	**88.91**	**2.29**×	**2.44**	**76.72**	**2.24**×
Token Recycling	2.83	69.65	1.84×	2.78	75.44	1.94×	2.88	66.17	1.93×
SAM-Decoding[T]	**3.03**	**85.73**	**2.27**×	**2.94**	**95.08**	**2.45**×	**3.23**	**87.93**	**2.57**×
EAGLE-2	4.36	90.14	2.38×	**5.13**	125.77	3.24×	4.15	82.61	2.41×
SAM-Decoding[E2]	**4.62**	**97.56**	**2.58**×	4.95	**130.28**	**3.35**×	**4.75**	**96.60**	**2.81**×

Table 1: Inference efficiency of different methods on Spec-Bench, HumanEval, and HAGRID, where each method was compared with the autoregressive decoding method provided in its environment.

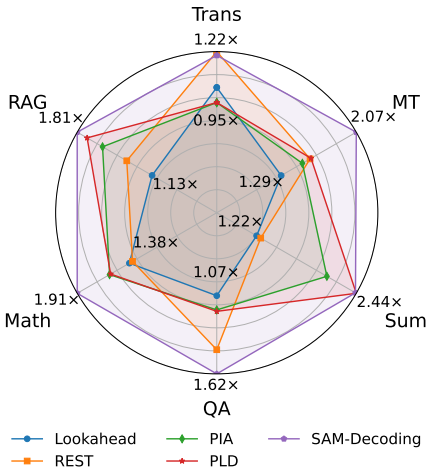

Figure 4: Speedup of SAM-Decoding compared to retrieval-based SD baselines on Spec-Bench.

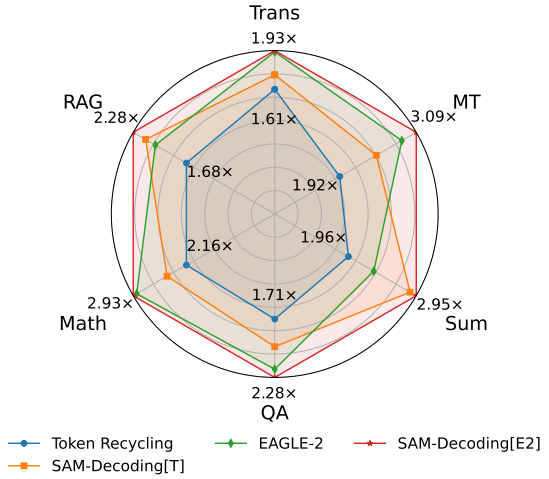

Figure 5: Speedup of SAM-Decoding combined with auxiliary SD methods compared to SD baselines on Spec-Bench.

EAGLE-2 (Li et al., 2024b), the model-free method Token Recycling (Luo et al., 2024), and the retrieval-based methods Lookahead Decoding (Fu et al., 2024), PIA (Zhao et al., 2024b), PLD (Saxena, 2023) and REST (He et al., 2024).

Metrics. We evaluated speculative decoding methods from the following aspects (Li et al., 2024c). **(1) Mean Number of Accepted Tokens (#MAT)**: The average number of tokens accepted per generation step. **(2) Throughput (Tokens/s)**: The average number of tokens generated per second. **(3) Speedup**: The wall-time speedup ratio of speculative decoding methods compared to autoregressive generation.

Experiment Setup. Experiments were run on a server with a 20-core CPU and an NVIDIA RTX A6000 GPU (48GB). We used PyTorch 2.3.0, Transformers 4.46.1, and CUDA 12.1. Models utilized float16 and greedy decoding with a batch size of 1. Hyperparameters l_{bias} and $l_{\text{threshold}}$ were set to 5, but l_{bias} was 0 without auxiliary methods. The draft size was 40 by default and 16 for code datasets. Default settings from the original papers were applied for auxiliary speculative decoding methods. For SAM-Decoding, we constructed a static suffix automaton based on the Vicuna-7B generation results on datasets Stanford-alpaca, python-code-instruction-18k, and GSM8k. Details of the static SAM construction process and its overhead are shown in Appendices A.2 and A.3. To enhance our model, we incorporated

	EAGLE-2	SAM-Decoding[E2]
Prefill	7.54(1.4%)	7.83(1.6%)
DraftGen	141.92(26.6%)	103.83(21.1%)
Decoding	346.7(62.5%)	326.34(65.7%)
Verification	13.34(2.5%)	32.98(7.0%)
Updating	23.84(4.5%)	17.97(3.7%)

Table 2: Fine-grained comparison of EAGLE-2 and SAM-Decoding[E2] in HumanEval

Method	Spec-Bench		
	#MAT	Tokens/s	Speedup
PLD	1.75	59.02	1.56×
SAM-Decoding	2.30	69.37	1.84×
w/o Static SAM	1.85	61.93	1.64×
w/o Dynamic SAM	1.63	50.37	1.33×

Table 3: The impact of different draft generation modules on inference speed.

two auxiliary approaches: the model-free Token Recycling and the model-based EAGLE-2. Here, SAM-Decoding[T], and SAM-Decoding[E2] denote the combinations of our base model with Token Recycling, and EAGLE-2, respectively.

Experiment Results. Experimental results on Spec-Bench, HumanEval and HAGRID when using Vicuna-7B-v1.3 are shown in Table 1. It can be seen that SAM-Decoding has higher speedups on all datasets compared to all the retrieval-based baselines, achieving speedup ratios of $1.84\times$, $2.29\times$, and $2.24\times$ on each of the three datasets. Combining SAM-Decoding with generation-based methods can further speed up processing. In the Spec-Bench and HAGRID datasets, integrating SAM-Decoding improves the inference speed of Token Recycling and EAGLE-2. For Spec-Bench, this enhancement raises the speedup ratios from $1.84\times$ and $2.38\times$ to $2.27\times$ and $2.58\times$ respectively. On the HAGRID dataset, the improvement is from $1.93\times$ and $2.41\times$ to $2.57\times$ and $2.81\times$. In the HumanEval dataset, the integration of SAM-Decoding speedup the Token Recycling, increasing its speedup ratios from $1.94\times$ to $2.45\times$. It also slightly improves the throughput of the model-based EAGLE-2, though decreases the mean accepted tokens. We additionally compare EAGLE-2 and SAM-Decoding[E2] at a finer granularity. The results are shown in Table 2. It can be seen that EAGLE-2 incurs a non-negligible time overhead when generating drafts, accounting for 26.6% of total inference time. In contrast, SAM generates drafts more quickly. Thus, although SAM-Decoding[E2]'s mean accepted tokens are slightly lower than those of EAGLE-2, the overall inference speed is faster due to faster draft generation.

In Figures 4 and 5, we further show the speedup of the different methods on each task of Spec-Bench. Compared to retrieval-based SD baselines, SAM-Decoding shows better inference speed across all tasks. Meanwhile, SAM-Decoding can effectively improve the inference speed of Token Recyling on each task, as well as the inference speed of EAGLE-2 on MT, SUM, and RAG tasks.

Finally, we investigated the impact of different modules within SAM-Decoding on inference speed. SAM-Decoding comprises two draft generation modules: the static suffix automaton and the dynamic suffix automaton. We measured the inference speed of SAM-Decoding after removing each of these two modules individually. The results are presented in Table 3. From the experimental results, it is clear that each module contributes to the acceleration of the decoding process. Notably, the dynamic suffix automaton has a significantly greater impact compared to the static suffix automaton. *This suggests that, in many cases, generating drafts from the dynamic context is more effective than retrieving drafts from a pre-existing text corpus*. However, although the main improvement comes from dynamic SAM, static SAM also proves effective. When used together, the enhancements are most pronounced, as we select dynamic and static SAM based on the token matching length. Notice that SAM-Decoding is slightly slower than REST when only static SAM is used according to Table 1, which we consider stemming from differences in the static corpus used and the implementation detail, and we analyze this in Appendix D. In addition to Vicuna-7B, we also conducted experiments on more models as shown in Figure 1. Please refer to Appendix B for more experimental results.

Ablation Experiments. To further understand the contributions of various components of SAM-Decoding and the influence of different hyperparameters on inference speed, we conducted a series of ablation studies. Firstly, we examined the effects of l_{bias} and $l_{\text{threshold}}$ on inference speed through a grid search. These parameters control the pref-

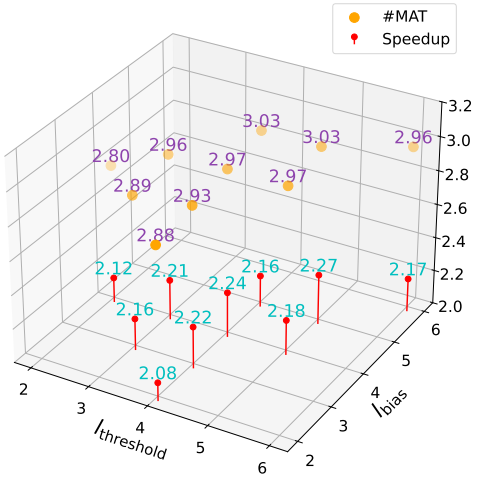

Figure 6: The speedup and mean accepted toknes of SAM-Decoding[T] under different l_{bias} and $l_{\text{threshold}}$.

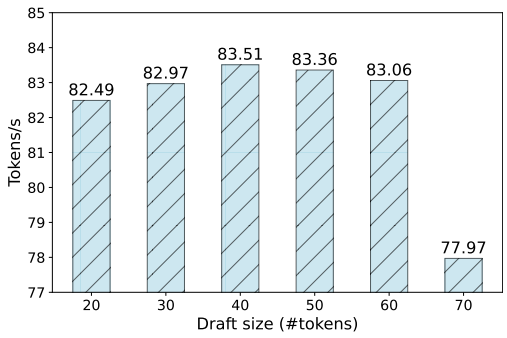

Figure 7: The throughput of SAM-Decoding[T] under different draft size.

erence for generating drafts from the current text over text corpus and the preference for using suffix automaton over the auxiliary SD method when creating drafts. The findings are summarized in Figure 6. *We observe that both the mean accepted tokens (MAT) and the speedup ratio increase with* l_{bias} *and* $l_{\text{threshold}}$ *before they equal 5 and then begin to decrease.*. Figure 7 illustrates the throughput of SAM-Decoding[T] at varying draft sizes. Increasing the draft size improves throughput when the draft size is less than 40. Above 40, throughput began to drop, and at 70, it dropped significantly. This is because increasing the draft size below the threshold reduces the number of generation rounds, thus improving efficiency. However, sizes above this threshold do not provide additional benefits, but rather strain the GPU compute density, which slows down inference. For more ablation experiment results, please refer to Appendix C.

5 Related Work

Speculative decoding is an approach that can significantly speed up large language models (LLMs) without compromising the quality of their outputs. The majority of speculative decoding techniques rely on smaller neural networks to create drafts during the inference process. These techniques are referred to as generation-based speculative decoding methods. Early implementations of model-based speculative decoding, such as those Speculative Decoding (Leviathan et al., 2023), primarily focused on generating draft sequences using pre-existing, smaller-scale LLMs. Subsequently, advancements like Medusa (Cai et al., 2024), SpecInfer (Miao et al., 2024), and EAGLE (Li et al., 2024c,b) introduced tree-based speculative methods (Du et al., 2024; Ankner et al., 2024; Chen et al., 2024b,a) and began to train additional model with special architecture tailored for speculative decoding. Token Recycling (Luo et al., 2024), on the other hand, utilizes the previously generated token distribution to generate drafts, becoming a specialized generation-based method. Additionally, beyond the additional model, research works also conducted on speculative decoding that relies either on the model itself (Kou et al., 2024; Yi et al., 2024) or on sub-layers within the LLM (Elhoushi et al., 2024; Liu et al., 2024). In contrast to generation-based methods, certain approaches focus on generating drafts through retrieval, utilizing n-gram matching, which we refer to as the retrieval-based method (Zhao et al., 2024a; Li et al., 2024a; Oliaro et al., 2024). Notable among these are Lookahead Decoding (Fu et al., 2024), PIA(Zhao et al., 2024b), PLD (Saxena, 2023) and REST (He et al., 2024).

6 Conclusion

In this work, we propose SAM-Decoding, a speculative decoding method via suffix automatons constructed from both generated text and text corpus. SAM-Decoding can efficiently retrieve drafts from retrieval sources, thereby accelerating inference. SAM-Decoding is also designed to seamlessly integrate with existing SD methods. Consequently, in scenarios where retrieval is not feasible, SAM-Decoding can adaptively switch to alternative methods for draft generation. Experimental results demonstrate that SAM-Decoding outperforms retrieval-based SD baselines. Meanwhile, when combined with state-of-the-art techniques, SAM-Decoding can significantly enhance their per-

formance in Multi-turn Conversation, Summarization, Retrieval-augmented Generation, and Context Q&A tasks.

7 Limitation

Firstly, when combining SAM-Decoding with other types of methods, we use a very heuristic approach, i.e., we choose different methods depending on the match length. This does not fully utilize the exact match lengths provided by the suffix automaton, so subsequently, we will try to train the classifier to select different decoding methods at each generation round.

Secondly, the performance of retrieval-based methods is highly correlated with the usage scenarios, and the existing datasets do not well reflect the performance of retrieval-based methods in real usage, so in the future, we also need to construct datasets that are more compatible with real scenarios to evaluate the performance of retrieval-based methods.

Finally, we developed the suffix automaton utilizing Python, which introduced unnecessary redundancy in its storage and concurrently compromised the efficiency of the automaton. Consequently, a pivotal direction for future work involves implementing the suffix automaton in more efficient programming languages to mitigate these issues. Moreover, alternative data structures such as full-text indexes (Ferragina et al., 2007) and compressed suffix trees (Shareghi et al., 2016) offer enhanced memory efficiency compared to the suffix automaton. Therefore, further investigation into whether these structures can facilitate more time- and memory-efficient draft retrieval is necessary.

Acknowledgments

This work is supported by the National Key Research & Develop Plan (2023YFF0725100), the National Natural Science Foundation of China (62322214, U23A20299, U24B20144, 62172424, 62276270, 62406164), and the Postdoctoral Fellowship Program of CPSF under Grant Number GZB20240358 and 2024M761680.

We also sincerely thank Cunxiao Du from the Sea AI Lab and Heming Xia from the Hong Kong Polytechnic University for their recognition of this work.

References

Zachary Ankner, Rishab Parthasarathy, Aniruddha Nrusimha, Christopher Rinard, Jonathan Ragan-Kelley, and William Brandon. 2024. Hydra: Sequentially-dependent draft heads for medusa decoding. *arXiv preprint arXiv:2402.05109*.

Anselm Blumer, Janet Blumer, Andrzej Ehrenfeucht, David Haussler, and Ross McConnell. 1984. Building the minimal dfa for the set of all subwords of a word on-line in linear time. In *Automata, Languages and Programming: 11th Colloquium Antwerp, Belgium, July 16–20, 1984 11*, pages 109–118. Springer.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. *Preprint*, arXiv:2005.14165.

Tianle Cai, Yuhong Li, Zhengyang Geng, Hongwu Peng, Jason D Lee, Deming Chen, and Tri Dao. 2024. Medusa: Simple llm inference acceleration framework with multiple decoding heads. *arXiv preprint arXiv:2401.10774*.

Jian Chen, Vashisth Tiwari, Ranajoy Sadhukhan, Zhuoming Chen, Jinyuan Shi, Ian En-Hsu Yen, and Beidi Chen. 2024a. Magicdec: Breaking the latency-throughput tradeoff for long context generation with speculative decoding. *arXiv preprint arXiv:2408.11049*.

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. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Zhuoming Chen, Avner May, Ruslan Svirschevski, Yuhsun Huang, Max Ryabinin, Zhihao Jia, and Beidi Chen. 2024b. Sequoia: Scalable, robust, and hardware-aware speculative decoding. *arXiv preprint arXiv:2402.12374*.

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

Cunxiao Du, Jing Jiang, Xu Yuanchen, Jiawei Wu, Sicheng Yu, Yongqi Li, Shenggui Li, Kai Xu, Liqiang Nie, Zhaopeng Tu, et al. 2024. Glide with a cape: A low-hassle method to accelerate speculative decoding. *arXiv preprint arXiv:2402.02082*.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, and Aiesha Letman et al. 2024. The llama 3 herd of models. *Preprint*, arXiv:2407.21783.

Mostafa Elhoushi, Akshat Shrivastava, Diana Liskovich, Basil Hosmer, Bram Wasti, Liangzhen Lai, Anas Mahmoud, Bilge Acun, Saurabh Agarwal, Ahmed Roman, et al. 2024. Layer skip: Enabling early exit inference and self-speculative decoding. *arXiv preprint arXiv:2404.16710*.

Paolo Ferragina, Giovanni Manzini, Veli Mäkinen, and Gonzalo Navarro. 2007. Compressed representations of sequences and full-text indexes. *ACM Trans. Algorithms*, 3(2):20–es.

Yichao Fu, Peter Bailis, Ion Stoica, and Hao Zhang. 2024. Break the sequential dependency of llm inference using lookahead decoding. *arXiv preprint arXiv:2402.02057*.

Zhenyu He, Zexuan Zhong, Tianle Cai, Jason Lee, and Di He. 2024. Rest: Retrieval-based speculative decoding. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 1582–1595.

Ehsan Kamalloo, Aref Jafari, Xinyu Zhang, Nandan Thakur, and Jimmy Lin. 2023. Hagrid: A human-llm collaborative dataset for generative information-seeking with attribution. *arXiv preprint arXiv:2307.16883*.

Vladimir Karpukhin, Barlas Oğuz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. 2020. Dense passage retrieval for open-domain question answering. *arXiv preprint arXiv:2004.04906*.

Siqi Kou, Lanxiang Hu, Zhezhi He, Zhijie Deng, and Hao Zhang. 2024. Cllms: Consistency large language models. *arXiv preprint arXiv:2403.00835*.

Tom Kwiatkowski, Jennimaria Palomaki, Olivia Redfield, Michael Collins, Ankur Parikh, Chris Alberti, Danielle Epstein, Illia Polosukhin, Jacob Devlin, Kenton Lee, Kristina Toutanova, Llion Jones, Matthew Kelcey, Ming-Wei Chang, Andrew M. Dai, Jakob Uszkoreit, Quoc Le, and Slav Petrov. 2019. Natural questions: A benchmark for question answering research. *Transactions of the Association for Computational Linguistics*, 7:452–466.

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

Minghan Li, Xilun Chen, Ari Holtzman, Beidi Chen, Jimmy Lin, Wen-tau Yih, and Xi Victoria Lin. 2024a. Nearest neighbor speculative decoding for llm generation and attribution. *arXiv preprint arXiv:2405.19325*.

Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. 2024b. Eagle-2: Faster inference of language models with dynamic draft trees. *arXiv preprint arXiv:2406.16858*.

Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. 2024c. Eagle: Speculative sampling requires rethinking feature uncertainty. *arXiv preprint arXiv:2401.15077*.

Jiahao Liu, Qifan Wang, Jingang Wang, and Xunliang Cai. 2024. Speculative decoding via early-exiting for faster llm inference with thompson sampling control mechanism. *arXiv preprint arXiv:2406.03853*.

Xianzhen Luo, Yixuan Wang, Qingfu Zhu, Zhiming Zhang, Xuanyu Zhang, Qing Yang, Dongliang Xu, and Wanxiang Che. 2024. Turning trash into treasure: Accelerating inference of large language models with token recycling. *arXiv preprint arXiv:2408.08696*.

Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Zeyu Wang, Zhengxin Zhang, Rae Ying Yee Wong, Alan Zhu, Lijie Yang, Xiaoxiang Shi, et al. 2024. Specinfer: Accelerating large language model serving with tree-based speculative inference and verification. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 932–949.

Ramesh Nallapati, Bowen Zhou, Caglar Gulcehre, Bing Xiang, et al. 2016. Abstractive text summarization using sequence-to-sequence rnns and beyond. *arXiv preprint arXiv:1602.06023*.

Gabriele Oliaro, Zhihao Jia, Daniel Campos, and Aurick Qiao. 2024. Suffixdecoding: A model-free approach to speeding up large language model inference. *arXiv preprint arXiv:2411.04975*.

Apoorv Saxena. 2023. Prompt lookup decoding.

Ehsan Shareghi, Matthias Petri, Gholamreza Haffari, and Trevor Cohn. 2016. Fast, small and exact: Infinite-order language modelling with compressed suffix trees. *Transactions of the Association for Computational Linguistics*, 4:477–490.

Heming Xia, Zhe Yang, Qingxiu Dong, Peiyi Wang, Yongqi Li, Tao Ge, Tianyu Liu, Wenjie Li, and Zhifang Sui. 2024. Unlocking efficiency in large language model inference: A comprehensive survey of speculative decoding. *arXiv preprint arXiv:2401.07851*.

An Yang, Baosong Yang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Zhou, Chengpeng Li, Chengyuan Li, Dayiheng Liu, Fei Huang, et al. 2024. Qwen2 technical report. *arXiv preprint arXiv:2407.10671*.

Hanling Yi, Feng Lin, Hongbin Li, Peiyang Ning, Xiaotian Yu, and Rong Xiao. 2024. Generation meets verification: Accelerating large language model inference with smart parallel auto-correct decoding. *arXiv preprint arXiv:2402.11809*.

Weilin Zhao, Yuxiang Huang, Xu Han, Chaojun Xiao, Zhiyuan Liu, and Maosong Sun. 2024a. Ouroboros: Speculative decoding with large model enhanced drafting. *arXiv preprint arXiv:2402.13720*.

Yao Zhao, Zhitian Xie, Chen Liang, Chenyi Zhuang, and Jinjie Gu. 2024b. Lookahead: An inference acceleration framework for large language model with lossless generation accuracy. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 6344–6355.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. 2023. Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in Neural Information Processing Systems*, 36:46595–46623.

A Suffix Automaton

A.1 Construction Process of Suffix Automaton

Algorithm 2 introduces the construction (BuildSAM) and expansion process (Expand) of Suffix Automaton, where the INIT_SAM function creates a suffix automaton that only contains the root node. For the root node, the **link** attribute value is None, the **next** attribute value is empty, the length attribute value is 0, and the min_endpos attribute value is 0. Meanwhile, Algorithm 3 shows the construction process of the top-k successors for each node of the static suffix automaton. Each node in the algorithm involves the variable, "freq", which represents the frequency of occurrence of the corresponding substring for each node, and can be initialized at the time of constructing the suffix automaton, i.e., "freq" is initialized to 1 for nodes generated by expansion, and "freq" is initialized to 0 for nodes generated based on cloning (line 15 and line 32 in Algorithm 2). For each node, the recording of variable 'freq' is used to estimate the probability of generating different successor tokens, thus determining the top-k successors.

A.2 Construction Detail

To construct static SAM, we converted instructions and inputs from the Stanford-Alpaca, Python-Code-Instruction-18K, and GSM8K datasets into model inputs using Vicuna's default template, as shown in Listing 1. We then generated responses for these inputs based on the Vicuna-7B-v1.3 model. Finally, the generated corpus—including the template, instructions, inputs, and responses—was used to construct our static SAM.

For the instructions and inputs, we selected Stanford-Alpaca due to its reputation for being lightweight and containing a diverse range of instructions. Since Stanford-Alpaca lacks math and coding instructions, we incorporated the Python-Code-Instruction-18K and GSM8K datasets to address these gaps.

For the responses, as the main experiments were based on the Vicuna series of models, we generated the responses for the instructions and inputs using the Vicuna-7B-v1.3 model to ensure alignment with the data distribution of the Vicuna series of models.

```
{
    "description": "Template used by
        Vicuna.",
    "prompt_input": "A chat between a
        curious user and an artificial
```

Corpus Size	#MAT	Tokens/s	Speedup
100%	2.30	69.37	1.84×
50%	2.24	68.35	1.81×
25%	2.21	67.90	1.80×
0% (w/o static SAM)	1.85	61.93	1.64×

Table 4: Inference speed of SAM-Decoding under different corpus size.

```
        intelligence assistant. The
        assistant gives helpful,
        detailed, and polite answers to
        the user's questions.\n\nUSER: {
        instruction}\n\n{input}\n\
        nASSISTANT:",
    "prompt_no_input": "A chat between a
         curious user and an artificial
        intelligence assistant. The
        assistant gives helpful,
        detailed, and polite answers to
        the user's questions.\n\nUSER: {
        instruction}\n\nASSISTANT:"
}
```

Listing 1: Vicuna's default template

A.3 Resources Usage and Effect of Static Corpus

The corpus we used to build the static SAM consists of 22,438,527 tokens, and the resulting static SAM takes up 1.5GB of memory. Each token needs approximately 72 bytes of storage space on average. In addition, it takes about 5 minutes to build static SAM and about 2 minutes to load static SAM from disk to memory.

Meanwhile, to further demonstrate how the corpus size of static SAM affects SAM-Decoding's inference speed, we tested subsets containing 50% and 25% of the data from our corpus. The results of these experiments are shown in Table 4. These results indicate that incorporating static SAM improves SAM-Decoding's inference speed. However, increasing the corpus size does not significantly further enhance performance.

In summary, currently, the memory costs associated with the corpus are low, while further increasing the corpus size does not significantly improve performance.

A.4 Drafting Process of Suffix Automaton

Algorithm 4 introduces the drafting process based on Prim's algorithm to find a maximum spanning tree. The insight of the algorithm is to approximate the probability distribution of the token based on the frequency and find the draft tree with the highest probability based on the maximum spanning tree algorithm. As shown in Algorithm 3, for static suffix automaton, we can offline maintain the frequency of occurrence of the corresponding substring for each node. Then, based on the recorded frequency for each node in the automaton, we can compute the top-k successor tokens and corresponding transition probabilities, where the transition probability is computed by dividing the frequency of occurrence of the target state by the frequency of occurrence of the current state. The time complexity of Prim's algorithm is $O(n \log n)$, where n denotes the draft size.

A.5 Time Complexity of Suffix Automaton

Consider a suffix automaton S with the initial state s_0, which corresponds to the root node of the automaton (representing the empty string). Suppose that state s_0 undergoes transitions through a sequence of L tokens $x = (x_1, x_2, \ldots, x_L)$:

$$s_i = \text{Transfer}(S, x_i, s_{i-1}), \quad i \in \{1, 2, \ldots, L\}.$$

We aim to demonstrate that the average time complexity of each state transition is $O(1)$, while the worst-case time complexity is $O(L)$.

First, let us define the matching length associated with the state s_i as l_i. Given that each state transition can increase the length of the match by at most 1, it follows that $0 \leq l_i \leq i$. Next, we introduce the concept of energy ϕ for each state s_i, defined as $\phi(s_i) = l_i$. Let c_i represent the time cost of the transition of the i-th state. We then define the amortized cost $\hat{c}_i$ as:

$$\hat{c}_i = c_i + \phi(s_i) - \phi(s_{i-1}).$$

We can now express the total amortized cost over all transitions as:

$$\begin{aligned}\sum_{i=1}^{L} \hat{c}_i &= \sum_{i=1}^{L} (c_i + \phi(s_i) - \phi(s_{i-1})) \\ &= \sum_{i=1}^{L} c_i + \phi(s_L) - \phi(s_0).\end{aligned}$$

Since $\phi(s_i) \geq 0$ and $\phi(s_0) = 0$, it follows that:

$$\sum_{i=1}^{L} \hat{c}_i \geq \sum_{i=1}^{L} c_i.$$

Next, we analyze the upper bound of $\hat{c}_i$. Each state transition involves moving through the **link**

edge zero or more times, followed by a move through the **next** edge. Transitioning through the **link** edge incurs a cost of 1 but decreases the potential by at least 1. Conversely, transitioning through the **next** edge incurs a cost of 1 and increases the potential by 1. Consequently, the amortized cost $\hat{c}_i$ is bounded above by 2, leading to:

$$\sum_{i=1}^{L} \hat{c}_i \leq 2L.$$

Thus, the average time complexity of state transitions is:

$$\frac{\sum_{i=1}^{L} c_i}{L} \leq \frac{2L}{L} = 2,$$

which is $O(1)$. In the worst case, a single operation may require up to l_i transitions through the **link** edge, followed by one transition through the **next** edge, resulting in a worst-case time complexity of $O(L)$.

B Additional Experiment Results

In this section, we first present the results of the experiment on Llama3-8B-instruct, Vicuna-13B-v1.3, and Vicuna-33B-v1.3.

Tables 5 and 6 present the speedup ratios of SAM-Decoding compared to baseline methods across the Spec-Bench, HumanEval, and HAGRID datasets, utilizing the Llama3-8B-instruct model. It can be seen that the inference speed of SAM-Decoding outperforms the strongest retrieval-based baseline PLD on all tasks. Meanwhile, SAM-Decoding, when paired with Token Recycling (SAM-Decoding[T]), brings speedups on all tasks. Specifically, SAM-Decoding enhances the speedup ratio of Token Recycling from 1.92×, 1.85×, and 1.82× to 2.09×, 2.04×, and 2.12× for Multi-turn Conversation, Summarization, and Retrieval-Augmented Generation tasks, respectively. This improvement raises the overall speedup ratio of token recycling in the Spec-Bench dataset from 1.91× to 2.05×. On the HumanEval and HAGRID datasets, SAM-Decoding increases the speedup ratio of Token Recycling from 1.99× and 2.17× to 2.16× and 2.30×, respectively. Furthermore, SAM-Decoding also amplifies the performance gains of EAGLE-2 in Multi-turn Conversation, Summarization, Retrieval-augmented Generation, Code Generation and Context Q&A tasks. The speedup ratios were increased from 2.08×, 1.85×, 1.87×, 2.37×, and 2.18× to 2.36×, 1.98×, 2.11×, 2.54× and 2.35× respectively.

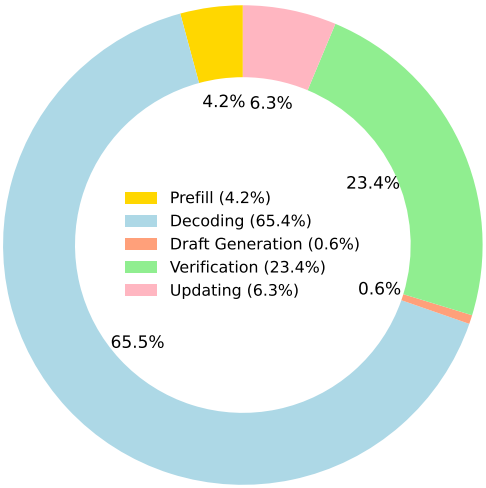

Figure 8: The percentage of inference time of different modules in SAM-Decoding.

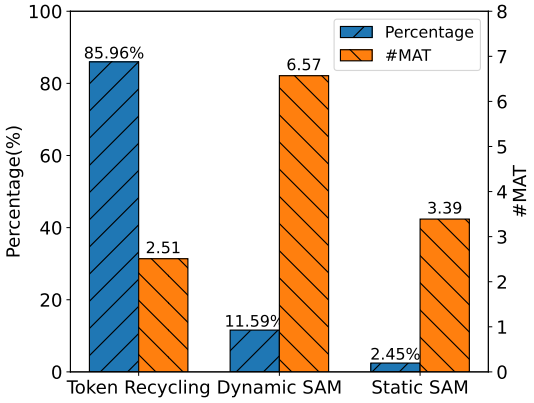

Figure 9: The percentage of usage and mean accept tokens of different draft modules.

Tables 7, 8, 9 and 10 present the speedup ratios of SAM-Decoding compared to baseline methods across the Spec-Bench, HumanEval, and HAGRID datasets, utilizing the Vicuna-13B-v1.3 and Vicuna-33B-v1.3. On both models, SAM-Decoding still has inference speed exceeding the retrieval-based baseline, while by combining Token Recycling and EAGLE-2 also further improves the inference speed of the model on the Multi-turn Conversation, Summarization, Retrieval-augmented Generation and Context Q&A tasks.

Then, we present additional experiments, including the percentage of inference time of different modules in the decoding process of SAM-Decoding, the percentage of drafts provided by different draft modules in SAM-Decoding, and the effect of different hyperparameters on the inference speed of SAM-Decoding for each task in Spec-Bench.

Model	Method	MT	Trans	Sum	QA	Math	RAG	#MAT	Tokens/s	Speedup
Llama3-8B	PLD	1.30×	1.12×	1.41×	1.03×	1.30×	1.53×	1.39	44.26	1.28×
	SAM-Decoding	1.59×	1.35×	1.50×	1.35×	1.54×	1.75×	1.72	52.35	1.51×
	Token Recycling	1.92×	1.88×	1.85×	1.75×	2.24×	1.82×	2.76	66.42	1.91×
	SAM-Decoding[T]	2.09×	1.93×	2.04×	1.82×	2.32×	2.12×	2.63	71.73	2.05×
	EAGLE-2	2.08×	1.95×	1.85×	1.80×	2.31×	1.87×	3.90	68.69	1.98×
	SAM-Decoding[E2]	2.36×	1.96×	1.98×	1.79×	2.32×	2.11×	3.92	72.47	2.08×

Table 5: Speedup of SAM-Decoding compared to the baselines on Spec-Bench.

Model	Method	HumanEval			HAGRID		
		#MAT	Tokens/s	Speedup	#MAT	Tokens/s	Speedup
Llama3-8B	PLD	1.30	42.39	1.18×	1.50	45.15	1.56×
	SAM-Decoding	2.06	64.38	1.79×	1.88	58.40	2.02×
	Token Recycling	2.93	71.49	1.99×	2.84	62.77	2.17×
	SAM-Decoding[T]	2.77	78.04	2.16×	2.70	66.76	2.30×
	EAGLE-2	4.74	85.58	2.37×	3.97	63.30	2.18×
	SAM-Decoding[E2]	4.76	91.50	2.54×	3.93	67.94	2.35×

Table 6: Speedup of SAM-Decoding compared to the baselines on HumanEval and HAGRID.

The inference process of SAM-Decoding is divided into five stages: prefill, draft generation, decoding, verification, and updating. During the prefill stage, the model processes the input prompt to establish an initial state. In the first draft generation stage, a draft is produced based on this initial state. The decoding phase consists of further processing of the draft by the model, i.e., feeding the draft into the LLM to obtain sampling results for each position. Next comes verification, where the correct parts of the draft are evaluated based on the information processed during the decoding stage. Finally, the update phase modifies the state of the model based on the valid parts of the draft. Figure 8 illustrates the proportion of time each stage consumes within the SAM-Decoding[T] process based on Spec-Bench. As shown, the decoding stage takes up the largest portion of time, accounting for 65.4% of the entire process. This is followed by the verification stage, which occupies 23.4% of the total time. The updating stage requires 6.3% of the time, whereas the draft generation stage contributes only 0.6% to the overall duration. Furthermore, the pre-fill stage comprises 4.2% of the total processing time.

Figure 9 shows the frequency of usage of different draft modules of SAM-Decoding[T] on Spec-Bench and the corresponding mean accept tokens. It can be seen that in 85.96% of the cases, due to insufficient matching length, we generate drafts based on the auxiliary method, corresponding to an average accept length of 2.51, while in the remaining 11.59% and 2.45% of the cases, the dynamic suffix automaton and static suffix automaton are used to generate drafts, corresponding to average accept lengths of 6.57 and 3.39, respectively. Table 12 further shows the frequency of use of different modules in each task and the corresponding mean accept tokens.

Finally, Table 11 shows the inference speed of different methods based on Vicuna-7B-v1.3 on NVIDIA A800 GPU. It can be seen that SAM-Decoding can still effectively combine Token Recycling and EAGLE-2 to achieve higher inference speed, which shows the effectiveness of our approach for different devices.

C Additional Ablation Experiments

In Figures 6 and 7, we illustrated the effect of $l_{\text{threshold}}$, l_{bias}, and draft size on the inference speed of SAM-Decoding[T], and here we further show the effect of these hyperparameters for each task in Spec-Bench. The experimental results are shown in Tables 14 and 15.

Model	Method	MT	Trans	Sum	QA	Math	RAG	#MAT	Tokens/s	Overall
Vicuna-13B	PLD	1.61×	1.10×	2.36×	1.11×	1.69×	1.80×	1.66	33.89	1.59×
	SAM-Decoding	2.08×	1.26×	2.23×	1.53×	2.09×	1.89×	2.19	39.24	1.84×
	Token Recycling	2.03×	1.84×	2.07×	1.83×	2.42×	1.84×	2.81	42.74	2.01×
	SAM-Decoding[T]	2.36×	1.80×	2.63×	1.83×	2.49×	2.22×	2.91	47.27	2.22×
	EAGLE-2	3.10×	2.15×	2.58×	2.38×	3.19×	2.33×	4.42	56.06	2.63×
	SAM-Decoding[E2]	3.27×	2.12×	2.89×	2.34×	3.12×	2.54×	4.51	57.88	2.72×

Table 7: Speedup of SAM-Decoding compared to the baselines on Spec-Bench.

Model	Method	HumanEval			HAGRID		
		#MAT	Tokens/s	Speedup	#MAT	Tokens/s	Speedup
Vicuna-13B	PLD	1.54	32.06	1.44×	1.90	43.38	2.15×
	SAM-Decoding	2.42	48.92	2.20×	2.21	41.93	2.08×
	Token Recycling	2.79	46.03	2.07×	2.90	40.97	2.03×
	SAM-Decoding[T]	2.79	50.87	2.28×	2.99	48.33	2.40×
	EAGLE-2	5.15	77.85	3.49×	4.24	52.28	2.59×
	SAM-Decoding[E2]	5.12	78.96	3.54×	4.41	56.17	2.78×

Table 8: Speedup of SAM-Decoding compared to the baselines on HumanEval and HAGRID.

D Comparison Between REST and SAM-Decoding with only Static SAM

In Table 3, we find that SAM-Decoding is slightly slower than REST when only static SAM is used (SAM-Decoding w/o Dynamic SAM) with the same mean accept tokens. We note that REST implements suffix arrays in C, whereas SAM-Decoding implements the SAM in Python. This difference results in a higher average per-operation overhead for SAM compared to suffix arrays, since Python is much slower than C. As a result, SAM-Decoding using only static SAM is currently slightly slower than REST. However, REST uses more corpus than SAM-Decoding. Specifically, the REST utilizes the ShareGPT dataset which is 10 times larger than the dataset employed by SAM-Decoding. Given this disparity, we augment our corpus with 10% of ShareGPT data and subsequently evaluate the SAM-Decoding inference speed on this enhanced dataset, which includes 38,521,232 tokens and 3 GB of memory for static SAM storage and is 1/5 the size of ShareGPT. Following this adjustment, SAM-Decoding demonstrated a mean accept token and inference speed that surpassed that of REST. In addition, it also indicates that static the suffix automaton have higher retrieval accuracy compared to the suffix arrays used by REST. The results are shown in Table 13.

Model	Method	MT	Trans	Sum	QA	Math	RAG	#MAT	Tokens/s	Overall
Vicuna-33B	PLD	1.50×	1.07×	2.06×	1.09×	1.59×	1.51×	1.65	13.33	1.46×
	SAM-Decoding	1.91×	1.25×	1.98×	1.48×	1.83×	1.66×	1.97	15.35	1.68×
	Token Recycling	2.10×	1.84×	2.19×	1.88×	2.42×	1.92×	2.70	18.80	2.06×
	SAM-Decoding[T]	2.31×	1.79×	2.53×	1.90×	2.48×	2.06×	2.68	19.87	2.18×
	EAGLE-2	3.29×	2.31×	2.73×	2.51×	3.65×	2.46×	4.06	25.86	2.83×
	SAM-Decoding[E2]	3.40×	2.25×	2.93×	2.43×	3.45×	2.54×	4.08	25.91	2.84×

Table 9: Speedup of SAM-Decoding compared to the baselines on Spec-Bench.

Model	Method	HumanEval			HAGRID		
		#MAT	Tokens/s	Speedup	#MAT	Tokens/s	Speedup
Vicuna-33B	PLD	1.58	14.18	1.51×	1.55	15.74	1.80×
	SAM-Decoding	2.05	19.08	2.03×	1.90	16.15	1.85×
	Token Recycling	2.64	19.64	2.09×	2.71	18.29	2.09×
	SAM-Decoding[T]	2.73	22.44	2.39×	2.60	19.74	2.26×
	EAGLE-2	3.53	28.18	3.00×	3.84	24.28	2.78×
	SAM-Decoding[E2]	3.61	29.56	3.14×	3.82	25.08	2.87×

Table 10: Speedup of SAM-Decoding compared to the baselines on HumanEval and HAGRID.

Model	Method	MT	Trans	Sum	QA	Math	RAG	#MAT	Tokens/s	Overall
Vicuna-7B	Token Recycling	2.08×	1.76×	1.97×	1.85×	2.35×	1.76×	2.82	98.39	1.96×
	SAM-Decoding[T]	2.62×	1.82×	2.92×	2.09×	2.60×	2.21×	3.02	119.21	2.38×
	EAGLE-2	2.66×	1.76×	2.18×	2.03×	2.63×	1.97×	4.34	110.56	2.21×
	SAM-Decoding[E2]	3.19×	1.97×	2.86×	2.28×	2.84×	2.32×	4.52	129.36	2.58×

Table 11: Speedup of SAM-Decoding on A800 GPU compared to the baselines on Spec-Bench.

	Token Recycling		Dynamic SAM		Static SAM	
	Percentage	#MAT	Percentage	#MAT	Percentage	#MAT
MT	86%	2.43	11%	7.44	3%	3.55
Trans	99%	2.18	1%	2.67	0%	NAN
Sum	81%	2.71	19%	8.24	0%	NAN
QA	92%	2.27	6%	11.22	2%	3.91
Math	79%	2.86	16%	4.01	5%	3.13
RAG	84%	2.57	13%	4.93	3%	2.93

Table 12: The frequency of use of different modules of SAM-Decoding[T] in each task of Spec-Bench.

Algorithm 2 Construction Process of Suffix Automaton

function Expand-State
  Input: suffix automaton S, link l, next nxt, length len, position p, frequency f
  $s = S$.expand_state() {A constructor}
  s.link $= l$
  s.next $= \mathrm{nxt}$
  s.length $= len$
  s.min_endpos $= p$
  s.freq $= f$
  Output: new state s
end function
function Expand
  Input: suffix automaton S, token t
  S.max_length $= S$.max_length $+ 1$
  $l = S$.max_length
  $c =$ Expand-State$(S, \mathrm{None}, \{\}, l, l, 1)$
  $p = S$.last
  while $p \neq$ None **and** $t \notin p$.next **do**
    p.next[t] $= c$
    $p = p$.link
  end while
  if $p =$ **None then**
    c.link $= S$.root
  else
    $q = p$.next[t]
    if p.length + 1 = q.length **then**
      c.link $= q$
    else
      $cl =$ Expand-State(S)
      cl.link $= q$.link, cl.next $= q$.next
      cl.length $= p$.length $+ 1$
      cl.min_endpos $= q$.min_endpos
      cl.freq $= 0$
      while $p \neq$ **None and** p.next[t] $= q$ **do**
        p.next[t] $= cl$
        $p = p$.link
      end while
      q.link $= c$.link $= cl$
    end if
  end if
  S.last $= c$
end function
function Build-SAM
  Input: token sequence s
  $S =$ INIT_SAM()
  for t **in** s **do**
    Expand(S, t)
  end for
  Output: suffix automaton S
end function

Algorithm 3 Construction Process of Top-k Successors and Transition Probabilities

function dfs
  Input: state s
  for $t_n, s_n \in s$.next **do**
    dfs(s_n)
    s.freq $= s$.freq $+ s_n$.freq
  end for
  s.topk_succs $= \mathrm{TopK}_{\mathrm{freq}}(s.\mathrm{next})$
  s.topk_prob $= []$
  for $t_n, s_n \in s$.topk_succ **do**
    s.topk_prob.append$(s_n$.freq$/s$.freq$)$
  end for
end function
function Init_topk
  Input: suffix automaton S
  dfs$(S$.root$)$
end function

Algorithm 4 Drafting via Prim's Algorithm

function Prim
  Input: suffix automaton S, state s, start token t
  $q =$ PriorityQueue()
  q.push$(\{1.0, s, t\})$
  tokens $= [],$ parents $= []$
  while q.size() > 0 **and** d.size() $\neq$ MAX_SIZE **do**
    $p,\ idx,\ s,\ t = q$.top()
    q.pop()
    tokens.append(t)
    parents.append(idx)
    for (t_n, s_n, p_n) **in** zip$(s$.topk_succ, s.topk_prob$)$ **do**
      $p_{\mathrm{new}} = p * p_n$
      $s_{\mathrm{new}} = s_n$
      $t_{\mathrm{new}} = t_n$
      q.push$(p_{\mathrm{new}}, \mathrm{len}(\mathrm{tokens}), s_{\mathrm{new}}, t_{new})$
    end for
  end while
  Output: draft tree $(\mathrm{tokens}, \mathrm{parents})$
end function

	Dataset	#MAT	Tokens/s
REST	ShareGPT	1.63	51.34
SAM-Decoding	Ours	1.63	50.37
	+ ShareGPT(10%)	1.68	51.98

Table 13: Comparison between REST and SAM-Decoding with only static SAM.

	$(l_{\text{threshold}}, l_{\text{bias}})$				
	(3, 3)	(5, 3)	(5, 5)	(5, 8)	(8, 8)
MT	2.36×	2.44×	2.49×	2.32×	2.39×
Trans	1.51×	1.72×	1.70×	1.57×	1.71×
Sum	2.92×	2.89×	2.94×	2.84×	2.70×
QA	1.79×	1.82×	1.97×	1.79×	1.90×
Math	2.06×	2.19×	2.40×	2.16×	2.36×
RAG	2.11×	2.11×	2.14×	2.08×	2.06×

Table 14: The effect of $l_{\text{threshold}}$, l_{bias} for each task in Spec-Bench.

	Draft Size		
	20	40	60
MT	2.48×	2.49×	2.43×
Trans	1.70×	1.70×	1.70×
Sum	2.86×	2.94×	2.93×
QA	1.96×	1.97×	1.96×
Math	2.39×	2.40×	2.31×
RAG	2.13×	2.14×	2.14×

Table 15: The effect of draft size for each task in Spec-Bench.