SAM Decoding: Speculative Decoding via Suffix Automaton

Anonymous ACL submission

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 incomplete retrieval resources, inefficient retrieval methods, and are constrained to certain domains. This paper presents a novel retrieval-based speculative decoding method that adapts suffix automaton (SAM) for efficient and accurate draft generation by utilizing common text corpus and dynamic text sequence. 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 anonymous 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 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. These methods can decrease inference latency substantially while maintaining decoding accuracy.

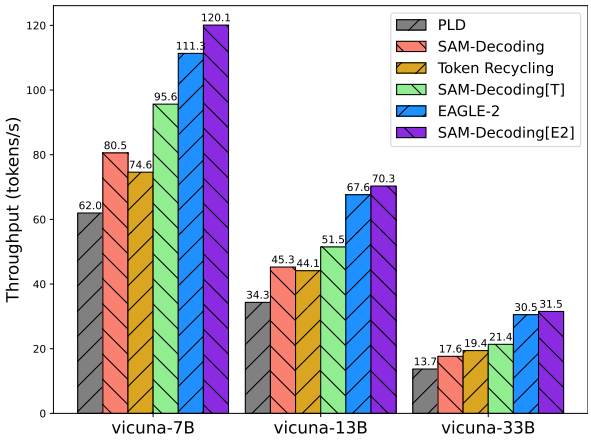

Figure 1: Throughput of Vicuna-7B, Vicuna-13B, Vicuna-33B on MT-Bench with A6000 GPU using PLD, Token Recycling (Luo et al., 2024), EAGLE-2, and SAM-Decoding, where PLD is the SOTA retrieval-based SD baseline.

Speculative methods can be categorized into model-based and model-free methods. Model-based methods need to carefully choose and train one or more small-sized draft models. For example, Medusa (Cai et al., 2024) utilizes multiple decoding heads to generate multiple future tokens while EAGLE-2 (Li et al., 2024a) leverages shallow Transformer layers to predict the next last hidden states and corresponding decoding tokens. 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 methods, aim to remedy this issue by generating draft tokens from text corpus or current text sequence.

However, current retrieval-based methods have 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 text corpus. **Secondly**, the retrieval techniques they use have efficiency limitations. PLD finds n-gram matching from current text sequence, but it has poor theoretical computational complexity and limited applicability to larger text corpus. REST uses suffixed arrays, which provides better complexity than PLD, but still not optimal complexity. **Thirdly**, retrieval-based methods are suitable for specialized domains (e.g., summarization and RAG), which are unable to bring a noticeable acceleration in other domains.

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 the common text corpus and the current text sequence 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 length 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 generalize our method, assuming the matching length of the longest suffix implying the quality of retrieval draft tokens, our method can be integrated with other types of speculative decoding methods, enabling more efficient text generation by deciding whether to adopt auxiliary decoding techniques.

Specifically, SAM-Decoding creates both a static suffix automaton for the text corpus and a dynamic suffix automaton for the current text sequence. The nodes of suffix automaton represent substrings in the text corpus or current sequence. The earliest position of each substring is recorded in each node. During generation, we can directly retrieve and filter drafts from the context using the matching positions and longest suffixes' matching length. After each generation step, the automaton is updated: static automaton nodes transition based on new tokens, while the dynamic automaton first expands its structure before node transitions.

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.). SAM-Decoding further achieves speedups of up to $1.3\times$ over alternative baselines on the code-generation benchmark like HumanEval. When combined with EAGLE-2 (Li et al., 2024a), 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.

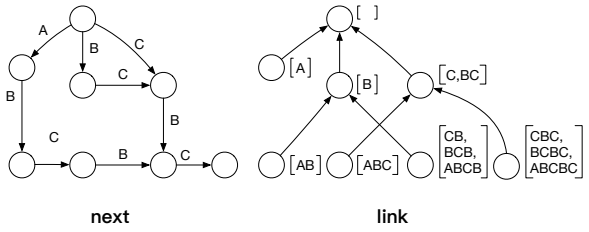

Figure 2: The suffix automaton corresponding to the string "ABCBC".

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 state transfer edges, **extension edges (next)** and **suffix link edges (link)**. A node in the automaton represents a state and 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 character, while 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 step of the generation with an average $O(1)$ time complexity.

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., $x_{\text{accept}} =$

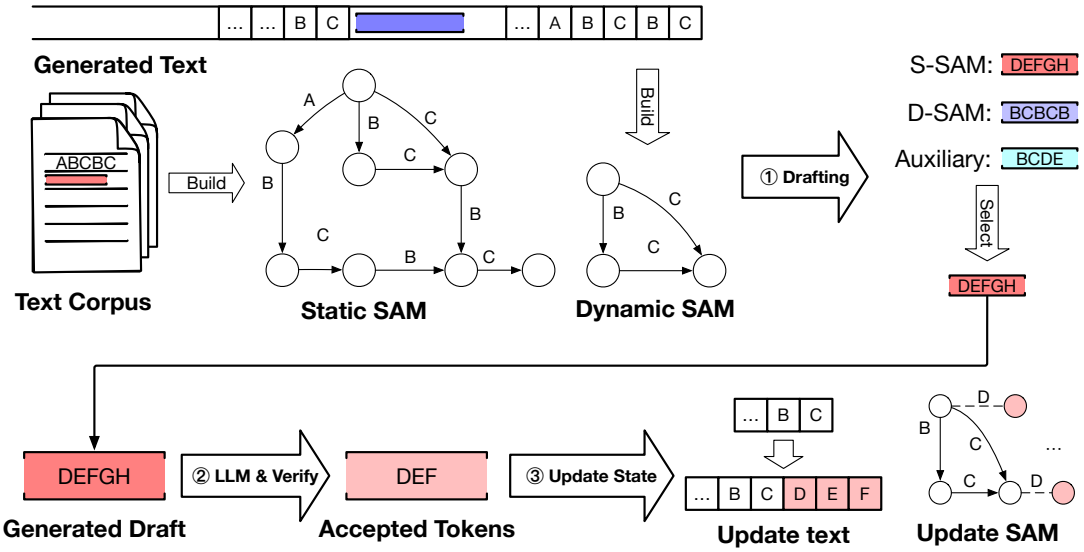

Figure 3: Overview of SAM-Decoding's workflow. In each round of generation, the suffix automaton matches the suffixes of the generating text and retrieves the draft from the text corpus and the generated text respectively according to the matching position. Our method can be combined with an auxiliary 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.

.

$(x_{t+1}, x_{t+2}, \ldots, x_{t+m})$, 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)** Subpar performance outside specialized domains (e.g., summarization and RAG tasks).

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. In what follows, we detail how SAM-Decoding can be integrated with both model-free and model-based methods. 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 common text corpus and the current text sequence (including user prompts and already generated tokens). Thus, we construct two types of suffix automaton: a **static suffix automaton** and a **dynamic suffix automaton**. For the text corpus, we pre-build a static suffix automaton offline, which is used for state matching during inference. For the current text sequence, we create and expand a dynamic suffix automaton incrementally as generation progresses and performing state matching concurrently.

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 reference string, static suffix automation can not be directly built using a text corpus. To this end, we concatenate multiple strings in the corpus by using special symbols like an End-of-Sentence (EOS)

token. We then construct a static suffix automaton for this concatenated string.

We have modified the suffix automaton for better draft generation. During each generation step, the current generated text sequence corresponds to a node in the automaton, representing the longest suffix match. At each node of the 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. The construction process of the suffix automaton is detailed in Appendix A.1.

For the static suffix automaton, based on the frequency of occurrence of different substrings, we additionally compute the top-k successor states (**topk_succ**) of each state, and subsequently use them to construct more complex tree drafts. Although computing the successor states requires significant computation, this can be done offline, eliminating the need to account for this time overhead in real-time processing.

3.2 Drafting with Suffix Automaton

We illustrate how to generate draft tokens efficiently based on the built 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. 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:

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

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:

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

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

For static suffix automaton, we construct a tree-structured draft by Prim's algorithm based on top-k successors, as detailed in Appendix A.2,

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

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**
    $s,\ l = s.\text{link},\ s.\text{link.length}$
  end while
  if $t \in s$.next **then**
    $s,\ l = s.\text{next}[t],\ l + 1$
  else
    $l = 0$
  end if
  Output: next state s, next matching length l
end function

In practical use, we track the longest-matched suffix 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. Our experimental findings indicate that drafts generated from the dynamic automaton often outperform those from the static text corpus. Consequently, we prioritize drafts from the dynamic automaton. We use the draft from the static automaton only if $l_1 > l_2 + l_{\text{bias}}$, where l_{bias} is a predefined constant.

The complete state transfer process of the suffix automaton is shown in Algorithm 1. Using amortized analysis, we can prove that the average complexity of state transfer is $O(1)$, with a worst-case time complexity of $O(L)$, where L is the length of the current generated text (C.f. proof in Appendix A.3). 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 $O(n^2 \log L)$. Here, n is the predefined maximum matching length, and L is the length of the current text or the concatenated texts in the corpus. In contrast, our proposed SAM-Decoding model 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.

3.3 Update of Suffix Automaton

After the draft is generated, we verify it using the large language model (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 state

of the suffix automaton based on these accepted tokens. For the static suffix automaton, we simply transfer the states according to Algorithm 1:

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

For the dynamic suffix automaton, we first transfer the matching state based on the accepted tokens and then expand the state. Let S_t denote the dynamic suffix automaton for the generated text $(x_1, x_2, \ldots, x_t)$. The process is as follows:

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

where the process of expanding the suffix automaton is detailed in Appendix A.1.

3.4 Adaptive Draft Selection

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 and builds a draft tree using breadth-first search, and it 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 Speculative Decoding 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), Mathmatical Reasoning (Math), and Retrieval-augmented Generation (RAG). In addition, HumanEval, and HARGID are used to evaluate the speed of decoding methods in Code Generation task and Context Q&A task, respectively.

Baselines. We considered the following baseline methods, including the model-based method EAGLE-2 (Li et al., 2024a), 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., 2024), PLD (Saxena, 2023) and REST (He et al., 2024).

Metrics. We evaluated speculative decoding methods from the following aspects (Li et al., 2024b)

- **Speedup Ratio**: The wall-time speedup ratio of speculative decoding methods compared to autoregressive generation methods.
- **Mean Accepted Tokens**: The average number of tokens accepted per generation step.
- **Throughput**: The average number of tokens generated per second.

Experiment Setup. We conducted experiments on a server equipped with a 20-core CPU and a single NVIDIA RTX A6000 GPU (48GB). The experiments were implemented using PyTorch 2.3.0, Transformers 4.46.1 and CUDA 12.1. For the models, we used the float16 data type and applied greedy decoding with a batch size of 1. Regarding hyperparameters, l_{bias} and $l_{\text{threshold}}$ were set to 5, but when there is no auxiliary method l_{bias} is set to 0. The size of the draft generated by the automaton was set to 40 by default, while for code datasets the size of the draft is set to 16. For the auxiliary

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 SAM-Decoding compared to the baselines on Spec-Bench, HumanEval, and HAGRID, where * indicates that the method was compared with the baseline provided in its environment.

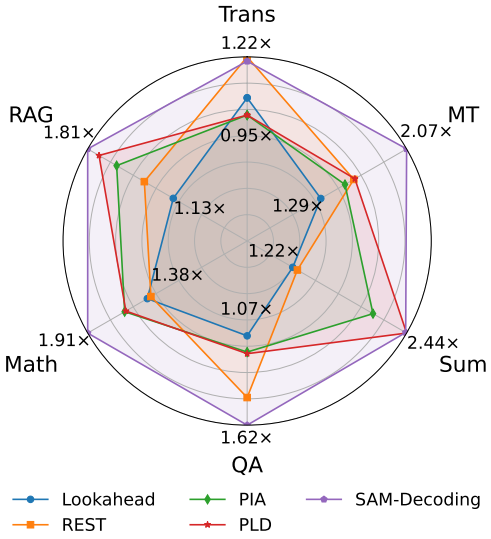

Figure 4: Relative speedup of SAM-Decoding compared to retrieval-based SD baselines on Spec-Bench.

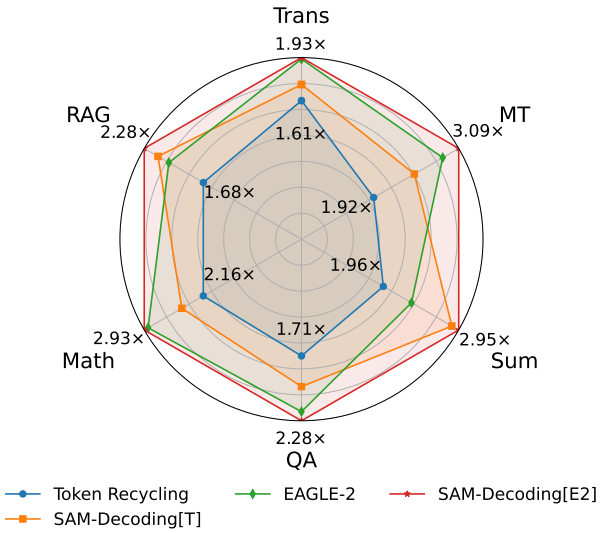

Figure 5: Relative speedup of SAM-Decoding compared to SD baselines on Spec-Bench when combined with auxiliary SD methods.

speculative decoding methods, we used the default configurations as described in their respective original papers.

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 GSK8k. To enhance our model, we incorporated 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 inference speedups on all datasets compared to retrieval-based baselines, achieving speedup ratios of $1.84\times$, $2.29\times$, and $2.24\times$ on each of the three datasets. Meanwhile, further speedups can be achieved by combining SAM-Decoding with other types of methods. On the Spec-Bench and HAGRID dataset, the inference speed of Token Recycling and EAGLE-2 can be further improved by combining SAM-Decoding. In Spec-Bench, the speedup ratios are improved from $1.84\times$, $2.38\times$ to $2.27\times$, $2.58\times$, respectively, whereas on HAGRID dataset, the

speedup ratios are improved from 1.93×, 2.41× to 2.57×, 2.81×. In the HumanEval dataset, the throughput of the model-based EAGLE-2 method changed slightly after integrating SAM-Decoding, due to the fact that the code generation task is less likely to copy the generated text during the generation process. Fortunately, SAM-Decoding can still speedup the model-free method Token Recycling, increasing its speedup ratios from 1.94× to 2.45×.

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 performance across all tasks. Meanwhile, in the Spec-Bench, Multi-turn Conversation, Summarization, and Retrieval-augmented Generation were identified as particularly amenable to retrieval techniques. The results indicate that integrating SAM-Decoding into existing method led to notable speed improvements. Specifically, for Token Recycling, the speedup ratio for the three tasks raised from 1.92×, 1.96×, and 1.68× to 2.48×, 2.86×, and 2.14×, respectively. For EAGLE-2, the speedup ratios raised from 2.87×, 2.33×, and 2.03× to 3.02×, 2.76×, and 2.23×, respectively.

In addition to Vicuna-7B, we also conducted experiments on more models. Figure 1 shows the throughput of Vicuna-7B, Vicuna-13B and Vicuna-33B on MT-bench using SAM-Decoding and other baseline SD methods. It can be seen that SAM-Decoding outperforms retrieval-based baselines on all models. Also, SAM-Decoding can further improve the inference speed of model-free and model-based SD methods by combining them with SAM-Decoding. For more experimental results, please refer to Appendix B.

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 preference for generating draft 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*. When the value of both parameters exceeds 5, these indicators begin to decline.

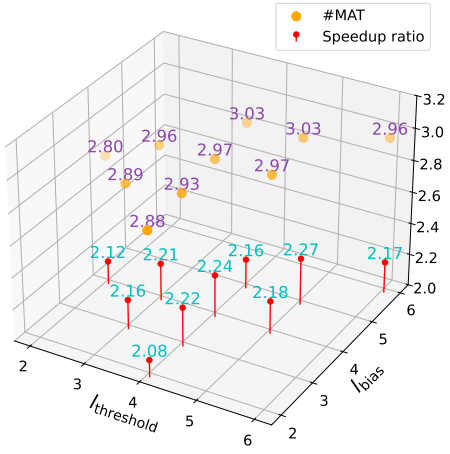

Figure 6: The speedup ratio and mean accepted toknes of SAM-Decoding[T] under different l_{bias} and $l_{\text{threshold}}$.

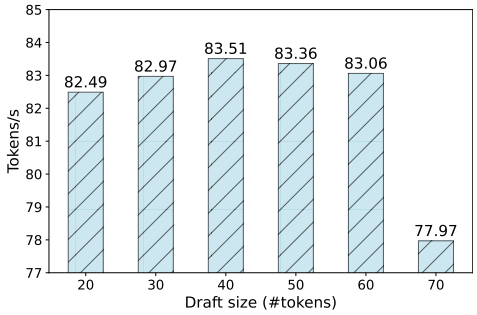

Figure 7: The throughput of SAM-Decoding[T] under different draft size.

Additionally, we investigated how the draft length utilized by SAM-Decoding affects inference speed. Figure 7 illustrates the throughput of SAM-Decoding[T] at varying draft sizes. *As the draft size increases, there is a positive trend in throughput until the draft size equals 40*. When the draft size exceeds 40, there is an observable decline in performance metrics, which becomes more significant as the draft size reaches 70. This phenomenon can be attributed to the fact that, for draft sizes below the average acceptance length, increasing the draft size reduces the number of rounds for generation, thereby enhancing efficiency. In contrast, once the draft size surpasses this threshold, further increases do not yield additional benefits and strain GPU capacity, thus slowing inference speed.

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

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

inference speed of SAM-Decoding after removing each of these two modules individually. The results are presented in Table 2. 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*. For more ablation experiment results, please refer to Appendix C.

5 Related Work

Speculative Decoding. 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 model-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., 2024b,a) introduced tree-based speculative methods and began the development of draft models tailored for speculative decoding.

In contrast to model-based methods, certain approaches focus on generating drafts through retrieval, utilizing n-gram matching, which we refer to the retrieval-based method. Notable among these are Lookahead Decoding (Fu et al., 2024), PIA(Zhao et al., 2024), PLD (Saxena, 2023) and REST (He et al., 2024). Token Recycling (Luo et al., 2024), on the other hand, utilizes the previously generated token distribution to generate drafts, becoming a model-free method different from the retrieval-based method.

Additionally, beyond the aforementioned methods, research also conducted on speculative decoding that relies either on the model itself (Kou et al., 2024) or on sub-models within the larger architecture (Elhoushi et al., 2024).

Efficient LLM Architecture. There is also work to improve the model's inference speed from the perspective of model structure. This part of the work includes model distillation, quantization and pruning. Model distillation (Sreenivas et al., 2024; Muralidharan et al., 2024) distills the knowledge of a large model into a small model thereby speeding up inference while maintaining the model's performance. Quantization (Frantar et al., 2022; Xiao et al., 2023; Lin et al., 2024; Liu et al., 2024; Ashkboos et al., 2024b) reduces the number of bits required to store parameters and reduces the data transmission time from HBM to on-chip memory during inference. Pruning (Frantar and Alistarh, 2023; Ashkboos et al., 2024a; Men et al., 2024; Chen et al., 2024; Hu et al., 2024; Sun et al., 2024; Zhang et al., 2024) is used to remove unimportant parameters in the model. For structured pruning, it can be combined with model distillation to train efficient small models, while semi-structured pruning can reduce the model's memory access and computing overhead and improve the inference speed by combining special hardware.

6 Conclusion

In this work, we propose SAM-Decoding, an 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 outperform retrieval-based SD baselines. Meanwhile, when combined with state-of-the-art techniques, SAM-Decoding can significantly enhance their performance in Multi-turn Conversation, Summarization, Retrieval-augmented Generation, and Context Q&A tasks.

7 Limitation

On the one hand, as a retrieval-based speculative decoding method, the performance of SAM-Decoding depends on the task type as well as the quality of the retrieval source. Currently, we have collected a text corpus based on the vicuna-7b generated results on Stanford-alpaca, GSM8k and python-instruct-18k. However, this corpus is still not diverse enough, and also the text in it may deviate from the text generated by other LLMs, which limits the performance of SAM-Decoding. Therefore, in the future we need to collect more specialized and diverse corpus for different types of tasks.

On the other hand, 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 classifier to select different decoding methods at each generate round.

Finally, 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.

References

Saleh Ashkboos, Maximilian L. Croci, Marcelo Gennari do Nascimento, Torsten Hoefler, and James Hensman. 2024a. Slicegpt: Compress large language models by deleting rows and columns. *Preprint*, arXiv:2401.15024.

Saleh Ashkboos, Amirkeivan Mohtashami, Maximilian L Croci, Bo Li, Pashmina Cameron, Martin Jaggi, Dan Alistarh, Torsten Hoefler, and James Hensman. 2024b. Quarot: Outlier-free 4-bit inference in rotated llms. *arXiv preprint arXiv:2404.00456*.

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*.

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*.

Xiaodong Chen, Yuxuan Hu, Jing Zhang, Yanling Wang, Cuiping Li, and Hong Chen. 2024. Streamlining redundant layers to compress large language models. *Preprint*, arXiv:2403.19135.

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*.

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*.

Elias Frantar and Dan Alistarh. 2023. Sparsegpt: Massive language models can be accurately pruned in one-shot. *Preprint*, arXiv:2301.00774.

Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. 2022. Gptq: Accurate post-training quantization for generative pre-trained transformers. *arXiv preprint arXiv:2210.17323*.

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.

Yuxuan Hu, Jing Zhang, Zhe Zhao, Chen Zhao, Xiaodong Chen, Cuiping Li, and Hong Chen. 2024. sp^3: Enhancing structured pruning via PCA projection. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 3150–3170, Bangkok, Thailand. Association for Computational Linguistics.

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.

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

Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han. 2024. Awq: Activation-aware weight quantization for on-device llm compression and acceleration. *Proceedings of Machine Learning and Systems*, 6:87–100.

Zechun Liu, Changsheng Zhao, Igor Fedorov, Bilge Soran, Dhruv Choudhary, Raghuraman Krishnamoorthi, Vikas Chandra, Yuandong Tian, and Tijmen Blankevoort. 2024. Spinquant–llm quantization with learned rotations. *arXiv preprint arXiv:2405.16406*.

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*.

Xin Men, Mingyu Xu, Qingyu Zhang, Bingning Wang, Hongyu Lin, Yaojie Lu, Xianpei Han, and Weipeng Chen. 2024. Shortgpt: Layers in large language models are more redundant than you expect. *Preprint*, arXiv:2403.03853.

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.

Saurav Muralidharan, Sharath Turuvekere Sreenivas, Raviraj Joshi, Marcin Chochowski, Mostofa Patwary, Mohammad Shoeybi, Bryan Catanzaro, Jan Kautz, and Pavlo Molchanov. 2024. Compact language models via pruning and knowledge distillation. *Preprint*, arXiv:2407.14679.

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*.

Apoorv Saxena. 2023. Prompt lookup decoding.

Sharath Turuvekere Sreenivas, Saurav Muralidharan, Raviraj Joshi, Marcin Chochowski, Mostofa Patwary, Mohammad Shoeybi, Bryan Catanzaro, Jan Kautz, and Pavlo Molchanov. 2024. Llm pruning and distillation in practice: The minitron approach. *arXiv preprint arXiv:2408.11796*.

Mingjie Sun, Zhuang Liu, Anna Bair, and J. Zico Kolter. 2024. A simple and effective pruning approach for large language models. *Preprint*, arXiv:2306.11695.

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*.

Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. 2023. Smoothquant: Accurate and efficient post-training quantization for large language models. In *International Conference on Machine Learning*, pages 38087–38099. PMLR.

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*.

Yingtao Zhang, Haoli Bai, Haokun Lin, Jialin Zhao, Lu Hou, and Carlo Vittorio Cannistraci. 2024. Plug-and-play: An efficient post-training pruning method for large language models. In *The Twelfth International Conference on Learning Representations*.

Yao Zhao, Zhitian Xie, Chen Liang, Chenyi Zhuang, and Jinjie Gu. 2024. 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 will create a suffix automaton that only contains the root node. For the root node, the **link** attribute value is -1, 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 static suffix automaton. Each node in the algorithm involves a new 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.

A.2 Drafting via Prim's Algorithm

Algorithm 4 introduces a drafting process based on Prim's algorithm to find a maximum spanning tree. For static suffix automata, we can offline maintain the frequency of occurrence of the corresponding substring for each node. Therefore, based on the recorded frequency for each node in the automaton, we can calculate the top-k successors and corresponding transition probabilities, where the transition probability is calculated by dividing the frequency of occurrence of the target state by the frequency of occurrence of the current state.

A.3 Time Complexity of State Transfer

In this section, we introduce the time complexity of state transfer of suffix automaton. Consider a suffix automaton S with 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 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 $0 \le l_i \le i$. 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:

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

$$= \sum_{i=1}^{L} c_i + \phi(s_L) - \phi(s_0).$$

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

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

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:

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

Thus, the average time complexity of state transitions is:

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

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 present the results of the experiment on Llama3-8B-instruct, Vicuna-13B-v1.3 and Vicuna-33B-v1.3.

Tables 3 and 4 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.

Tables 5, 6, 7 and 8 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.

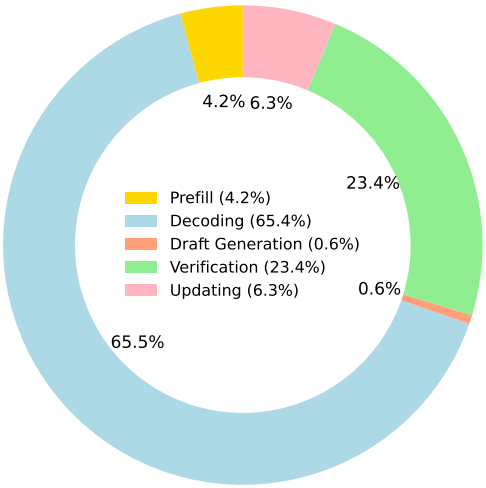

Figure 8: the percentage of inference time of different modules in SAM-Decoding.

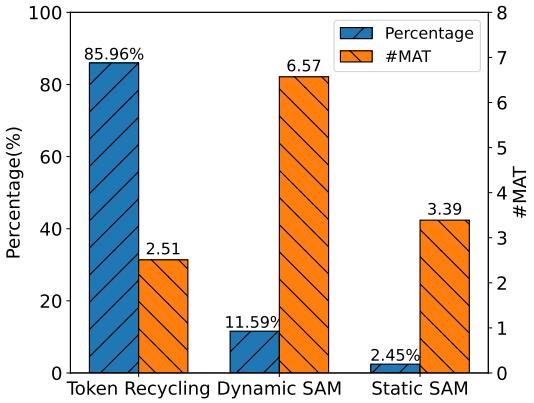

Figure 9: the percentage of usage and mean accept tokens of different draft modules.

C Additional Ablation Experiments

In this section, we present additional ablation experiments, including the percentage of inference time of different modules in the decoding process of SAM-Decoding, and the percentage of drafts provided by different draft modules in SAM-Decoding.

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 stage involves the model further processing this draft. 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

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 3: 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 4: Speedup of SAM-Decoding compared to the baselines on HumanEval and HAGRID.

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. Additionally, the prefill stage comprises 4.2% of the total processing time.

Figure 9 shows the usage frequency of different draft modules of SAM-Decoding[T] on Spec-Bench and the corresponding average draft accept length. 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.

Finally, Table 9 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.

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 5: 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 6: 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-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 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-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 8: 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 9: Speedup of SAM-Decoding on A800 GPU compared to the baselines on Spec-Bench.

Algorithm 2 Construction Process of Suffix Automaton

function Expand-State
Input: suffix automaton S, link l, next n, length len, position p
$s = S$.expand_state()
s.link $= l$
s.next $= n$
s.length $= len$
s.min_endpos $= p$
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$(S, -1, \{\}, l, l)$
$p = S$.last
while $p \neq -1$ **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$(S, , -1, \{\}, -1, -1)$
cl.link $= q$.link
cl.next $= q$.next
cl.length $= p$.length $+ 1$
cl.min_endpos $= q$.min_endpos
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 $=$ TopK$_{\text{freq}}(s$.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$(\{1.0, s, t\})$
$d =$ []
while q.size() > 0
and d.size() $\neq$ MAX_SIZE **do**
$p,\ s,\ t = q$.top()
q.pop()
d.append(t)
for (t_n, s_n, p_n) **in**
zip$(s$.topk_succ, s.topk_prob$)$ **do**
$p_{\text{new}} = p * p_n$
$s_{\text{new}} = s_n$
$t_{\text{new}} = t_n$
q.push$(p_{\text{new}}, s_{\text{new}}, t_{new})$
end for
end while
Output: draft tree d
end function