An Improved Data Structure for Left-Right Maximal Generic Words Problem

Yuta Fujishige
Department of Informatics, Kyushu University, Japan
yuta.fujishige@inf.kyushu-u.ac.jp

Yuto Nakashima
Department of Informatics, Kyushu University, Japan
yuto.nakashima@inf.kyushu-u.ac.jp

Shunsuke Inenaga
Department of Informatics, Kyushu University, Japan
inenaga@inf.kyushu-u.ac.jp

Hideo Bannai
Department of Informatics, Kyushu University, Japan
bannai@inf.kyushu-u.ac.jp

Masayuki Takeda
Department of Informatics, Kyushu University, Japan
takeda@inf.kyushu-u.ac.jp

Abstract

For a set D of documents and a positive integer d, a string w is said to be *d-left-right maximal*, if (1) w occurs in at least d documents in D, and (2) any proper superstring of w occurs in less than d documents. The *left-right-maximal generic words problem* is, given a set D of documents, to preprocess D so that for any string p and for any positive integer d, all the superstrings of p that are d-left-right maximal can be answered quickly. In this paper, we present an $O(n \log m)$ space data structure (in words) which answers queries in $O(|p| + o \log \log m)$ time, where n is the total length of documents in D, m is the number of documents in D and o is the number of outputs. Our solution improves the previous one by Nishimoto et al. (PSC 2015), which uses an $O(n \log n)$ space data structure answering queries in $O(|p| + r \cdot \log n + o \cdot \log^2 n)$ time, where r is the number of right-extensions q of p occurring in at least d documents such that any proper right extension of q occurs in less than d documents.

2012 ACM Subject Classification Mathematics of computing → Combinatorial algorithms; Mathematics of computing → Combinatorics on words

Keywords and phrases generic words, suffix trees, string processing algorithms

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2019.40

Funding *Yuto Nakashima*: Supported by JSPS KAKENHI Grant Number JP18K18002.
Shunsuke Inenaga: Supported by JSPS KAKENHI Grant Number JP17H01697.
Hideo Bannai: Supported by JSPS KAKENHI Grant Number JP16H02783.
Masayuki Takeda: Supported by JSPS KAKENHI Grant Number JP18H04098.

1 Introduction

String Data Mining is an important research area which has received special attention. One of the fundamental tasks in this area is the frequent pattern mining, the aim of which is to find patterns occurring in at least d documents in D for a given collection D of documents and a given threshold d, where the patterns are drawn from a fixed hypothesis space. The task is useful not only in extracting patterns which characterize the documents in D, but also in enumerating candidates for the most classificatory pattern that separates two given sets of

strings. The hypothesis space varies depending upon users' particular interest or purpose, and is ranging from the substring patterns to the VLDC patterns. Frequent substring patterns are often referred to as *generic words*. The generic words mining problem (or the frequent substring pattern mining problem) has a wide variety of applications, e.g., Computational Biology, Text mining, and Text Classification [5, 2, 3].

One interesting variant of the generic words mining problem is the *right maximal generic words problem*, formulated by Kucherov et al. [5]. In this variant, a pattern p is given as additional input, which limits the outputs to the right extensions of p. Moreover, the outputs are limited to the *maximal* ones. Formally, the problem is to preprocess D so that, for any pattern p and for any threshold d, all right extensions of p that are d-right maximal can be computed efficiently, where a string w is said to be *d-right maximal* if x occurs in at least d documents but xa occurs in less than d documents for any character a. They presented in [5] an $O(n)$-size data structure which answers queries in $O(|p| + r)$ time, where n is the total length of strings in D and r is the number of outputs. Later, Biswas et al. [2] developed a succinct data structure of size $n \log |\Sigma| + o(n \log |\Sigma|) + O(n)$ bits of space, which answers queries in $O(|p| + \log \log n + r)$ time.

As a generalization, Nishimoto et al. [7] defined the *left-right-maximal generic word problem*. In this problem, all superstrings of p that are d-left-right maximal should be answered, where a string w is said to be *d-left-right maximal* if x has a document frequency $\geq d$ but xa and ax respectively have a document frequency $< d$ for any character a.

One naive solution to this problem is to compute the sets M_d of d-left-right maximal strings for $d = 1, \ldots, m$, where m is the number of documents in D and then apply the optimal algorithm of Muthukrishnan [6] for the document listing problem, regarding M_d as input document collection. The query time is $O(|p| + o)$ time, where o is the number of outputs. The space requirement is $O(n^2 \log m)$ since the Muthukrishnan algorithm uses the (generalized) suffix tree of input document collection and the size of suffix tree for M_d can be shown to be $O(n^2/d)$ for every $d = 1, \ldots, m$. The $O(n^2 \log m)$ space requirement is, however, impractical when dealing with a large-scale document collection.

In [7] Nishomoto et al. presented an $O(n \log n)$-space data structure which answers queries in $O(|p| + r \log n + o \log^2 n)$ time, where r is the number of d-right-maximal strings that subsume p as a prefix. The factor $O(r \log n)$ is for computing the d-right-maximal right extensions of p, which are required for computing d-left-right-maximal extensions of p in their method.

In this paper, we address the left-right-maximal generic word problem and propose an $O(n \log m)$-space data structure with query time $O(|p| + o \log \log m)$. The data structure outperforms the previous work by Nishimoto et al. [7] both in the query time and in the space requirement.

Our method uses the suffix trees of M_d for $d = 1, \ldots, m$. For a string set $S = \{w_1, \ldots, w_\ell\}$, Usually, "the suffix tree of S" means the suffix tree of $\{w_1\$_1, \ldots, w_\ell\$_\ell\}$ with ℓ distinct endmarkers $\$_1, \ldots, \$_\ell$, or the suffix tree of $S\$ = \{w_1\$, \ldots, w_\ell\$\}$ with a single endmarker $\$$. In both cases, the size of suffix tree is proportional to the total length of the strings in S. The total size of suffix trees of $M_d\$$ for $d = 1, \ldots, m$ is $O(nm)$, where n is the total length of D. Our idea in reducing the space requirement is to replace the suffix tree of $M_d\$$ with the suffix tree of M_d. Removing the endmarker successfully reduces the $O(nm)$ total size of the suffix trees to $O(n \log m)$, with a small sacrifice of query time.

2 Preliminaries

2.1 Strings

Let Σ be an alphabet, that is, a nonempty, finite set of characters. Throughout this paper, we assume that Σ is an ordered alphabet of constant size. A *string* over Σ is a finite sequence of characters from Σ. Let Σ^* denote the set of strings over Σ. The *length* of a string w is the number of characters in w and denoted by $|w|$. The string of length 0 is called the *empty string* and denoted by ε. Let $\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$. The i-th character of a string w is denoted by $w[i]$ for $1 \leq i \leq |w|$. Strings x, y, and, z are, respectively, said to be a *prefix*, *substring*, and *suffix* of string $w = xyz$. The substring of a string w that begins at position i and ends at position j is denoted by $w[i..j]$ for $1 \leq i \leq j \leq |w|$. That is, $w[i..j] = w[i] \cdots w[j]$. For convenience, let $w[i..j] = \varepsilon$ for $i > j$. We use $w[..j]$ and $w[i..]$ as abbreviations of $w[1..j]$ and $w[i..|w|]$. Let $Pre(w)$, $Sub(w)$ and $Suf(w)$ denote the sets of prefixes, substrings, and suffixes of a string w, respectively. Let $Pre(S) = \bigcup_{w \in S} Pre(w)$, $Sub(S) = \bigcup_{w \in S} Sub(w)$ and $Suf(S) = \bigcup_{w \in S} Suf(w)$ for any set S of strings. The *reversal* of a string w, denoted by w^R, is defined to be $w[|w|] \dots w[1]$. Let $S^R = \{w^R \mid w \in S\}$ for any set S of strings.

The *longest repeating suffix* of a string x is the longest suffix of x that occurs elsewhere in x. Let $LRS(x)$ denote the length of the longest repeating suffix of x. We note that any suffix of x longer than $LRS(x)$ occurs only once in x.

2.2 d-left-right maximality of strings

Let D be a set of documents (strings). The *document frequency* of a string x in D, denoted by $df_D(x)$, is defined to be the number of documents in D that contain x as a substring. We write $df(x)$ instead of $df_D(x)$ when D is clear from the context.

A string x is said to be *d-left maximal* w.r.t. D if $df(x) \geq d$ and $df(ax) < d$ for all $a \in \Sigma$, and said to be *d-right maximal* w.r.t. D if $df(x) \geq d$ and $df(xa) < d$ for all $a \in \Sigma$. A string x is said to be *d-left-right maximal* w.r.t. D if it is d-left maximal and d-right maximal w.r.t. D. Let M_d denote the sets of d-left-right maximal strings w.r.t. D.

▶ **Example 1.** For $D = \{\texttt{aaabaabaaa}, \texttt{aaabaabbba}, \texttt{aababababaa}, \texttt{abaababbba}\}$, the sets of d-left-right maximal strings for $d = 1, 2, 3, 4$ are as follows: $M_1 = D$, $M_2 = \{\texttt{aaabaab}, \texttt{aabab}, \texttt{abaaba}, \texttt{ababb}, \texttt{abbba}\}$, $M_3 = \{\texttt{aaba}, \texttt{abaab}, \texttt{abb}, \texttt{bba}\}$ and $M_4 = \{\texttt{aaba}, \texttt{baa}\}$.

▶ **Lemma 2** ([5]). *For any set D of strings with total length n, the number of d-right maximal strings w.r.t. D is $O(n/d)$.*

▶ **Lemma 3.** *For any string y the following statements hold.*

1. *Let z be the shortest string such that $yz \in Suf(M_d)$. If $xyz \in M_d$ for some string x, then xy is d-left maximal.*
2. *Let x be the shortest string such that $xy \in Pre(M_d)$. If $xyz \in M_d$ for some string z, then yz is d-right maximal.*

Proof. It suffices to give proof only for the first statement. Suppose to the contrary that xy is not d-left maximal. Then, $df(xy) \geq df(xyz) \geq d$, and there exists some $\alpha \in \Sigma^+$ such that αxy is d-left maximal. Since xyz is d-maximal, $df(\alpha xyz) < d$. Furthermore, since $df(\alpha xy) \geq d$, there exists a prefix z' of z such that $\alpha xyz'$ is d-maximal and $|z'| < |z|$. This implies $yz' \in Suf(M_d)$ and contradicts that z is the shortest such string. Therefore, xy must be d-left maximal. ◀

2.3 Suffix trees

Let $S = \{w_1, \ldots, w_\ell\}$ be a set of nonempty strings with total length n. The suffix tree [8] of S, denoted by $ST(S)$, is a path-compressed trie which represents all suffixes of S. More formally, $ST(S)$ is an edge-labeled rooted tree such that (1) Every internal node is branching; (2) The out-going edges of every internal node begin with mutually distinct characters; (3) Each edge is labeled by a non-empty substring of S; (4) For each suffix s of S, there is a unique path from the root which spells out s and the path possibly ends on an edge; (5) Each path from the root to a leaf spells out a suffix of S. It follows from the definition of $ST(S)$ that the numbers of nodes and edges in $ST(S)$ are $O(n)$, respectively. By representing every edge label x by a triple (i, j, k) of integers such that $x = w_k[i..j]$, $ST(S)$ can be represented in $O(n)$ space. The *size* of suffix tree $ST(S)$ is defined to be the number of nodes and is denoted by $|ST(S)|$.

A node v of $ST(S)$ is said to *represent* a string x if the path from the root to v spells out x. For a substring x of S, the *locus* of x in $ST(S)$ is defined to be the highest node v that represents a right extension of x. A string x is said to be *explicit* in $ST(S)$ if there exists a node v of $ST(S)$ that represents x and *implicit* otherwise.

In this paper, we properly use the suffix trees of the following three types to suit its use.

1. $ST(\overline{S})$ where $\overline{S} = \{w_1\$_1, \ldots, w_\ell\$_\ell\}$ and $\$_1, \ldots, \$_\ell$ are mutually distinct endmarkers not in Σ.
2. $ST(S\$)$ where $\$$ is an endmarker not in Σ.
3. $ST(S)$ without endmarker.

The above suffix trees are all capable of determining whether $x \in \mathit{Sub}(S)$ for any $x \in \Sigma^+$. $ST(S)$ cannot distinguish the elements of $\mathit{Suf}(S)$ from those of $\mathit{Sub}(S)$ whereas $ST(S\$)$ and $ST(\overline{S})$ can determine whether $x \in \mathit{Suf}(S)$ for any $x \in \mathit{Sub}(S)$. In addition, $ST(\overline{S})$ can determine the set of indices k such that $x \in \mathit{Suf}(w_k)$. It is easy to see that:

▶ **Lemma 4.** *$|ST(\overline{S})| \geq |ST(S\$)| \geq |ST(S)|$ for any set S of strings.*

2.4 Tools

Let x be a fixed string over $A = \{1, \ldots, \sigma\}$. The Rank query $\mathit{rank}_x(a, i)$ returns the number of occurrences of $a \in A$ in the prefix $x[..i]$ of x, and the Select query $\mathit{select}_x(a, j)$ returns the position of j-th occurrence of $a \in A$ in x.

▶ **Lemma 5** ([4])**.** *There is an $O(|x|)$ space data structure that answers Rank/Select queries in $O(\log\log\sigma)$ time.*

Let T be an ordered tree with n nodes and with function val that maps the nodes to the integers. The find-less-than (FLT) query on tree T is, given a threshold τ and a node v of T, to enumerate the descendants u of v with $\mathit{val}(u) < \tau$.

▶ **Lemma 6.** *We can build from T an $O(n)$ space data structure in $O(n)$ time that answers FLT queries in $O(out)$ time, where out is the number of outputs.*

Proof. Let $v_1, \ldots, v_n$ be the nodes T in the preorder. Let B be an array such that $B[i] = \mathit{val}(v_i)$ for all $i \in [1..n]$. Then, the problem of FLT queries on tree can be reduced to the problem of FLT queries on array B defined as follows:

> Given a threshold τ and a subinterval $[i..j]$ of $[1..n]$, enumerate the indices k in $[i..j]$ such that $\mathit{val}(B[k]) < \tau$.

FLT queries on array B of size n can be answered in linear time proportional to the number of outputs, by repeated use of the Range Minimum Query (see [6]). ◀

2.5 Computation model

Our model of computation is the word RAM: We shall assume that the computer word size is at least $\lceil \log_2 n \rceil$, and hence, standard operations on values representing lengths and positions of strings can be manipulated in constant time. Space complexities will be determined by the number of computer words (not bits).

3 Main Result and Algorithm Outline

3.1 Main result

Our problem is formulated as follows.

▶ **Problem 7.**
To-preprocess: *A subset* $D = \{w_1, \ldots, w_m\}$ *of* Σ^+.
Query: *A string* $p \in \Sigma^+$ *and an integer* $d \in [1..m]$.
Answer: *The strings in* $\Sigma^* p \Sigma^* \cap M_d$.

One naive solution to the problem would be to apply the optimal algorithm of Muthukrishnan et al. [6] for the document listing problem regarding M_d as input document collection. This solution requires space proportional to the total size of suffix trees $ST(\overline{M_d})$ for $d = 1, \ldots m$.

▶ **Lemma 8.** *The suffix trees* $ST(M_d\$)$ *and* $ST(M_d)$ *are of size* $O(n)$ *for any* $d = 1, \ldots, m$, *and the suffix tree* $ST(\overline{M_d})$ *is of size* $O(n^2/d)$ *for any* $d = 1, \ldots, m$.

Proof. First, we show that $ST(M_d\$)$ has $O(n)$ leaves. Let v be any leaf of $ST(M_d\$)$, and let $x\$$ be the string represented by v. There is a string α such that $\alpha x \in M_d$. Assume, for a contradiction, that x is implicit in $ST(\overline{D})$. Then, there uniquely exists a character a such that every occurrence of x in the strings of $\overline{D}$ is followed by a. This contradicts $\alpha x \in M_d$. Hence x is explicit in $ST(\overline{D})$. The number of leaves of $ST(M_d\$)$ is not greater than the number of nodes of $ST(\overline{D})$, which is $O(n)$. By Lemma 4, $ST(M_d\$)$ and $ST(M_d)$ are of size $O(n)$. Next, we prove that $ST(\overline{M_d})$ has $O(n^2/d)$ leaves. Let v be any leaf of $ST(\overline{M_d})$, and let $x\$_i$ be the string represented by v. As the previous discussion, x is explicit in $ST(\overline{D})$. There are $|M_d|$ endmarkers $\$_j$ in $\overline{M_d}$, and by Lemma 2 we have $|M_d| = O(n/d)$. Hence the number of leaves of $ST(\overline{M_d})$ is not greater than $O(n/d)$ times the number of nodes of $ST(\overline{D})$, which is $O(n^2/d)$. ◀

The naive solution answers queries in $O(|p| + o)$ time using $O(n^2 \log m)$ space, where o is the number of outputs. The $O(n^2 \log m)$ space requirement is, however, impractical for dealing with a large-scale document set.

Our solution reduces the $O(n^2 \log m)$ space requirement to $O(n \log m)$ with a little sacrifice in query response time.

▶ **Theorem 9.** *There exists an* $O(n \log m)$ *space data structure for Problem 7 which answers queries in* $O(|p| + o \log \log m)$ *time, where* o *is the number of outputs.*

3.2 Algorithm outline

Our task is, given a string $p \in \Sigma^+$, to enumerate the strings $\alpha p \beta$ in M_d with $\alpha, \beta \in \Sigma^*$. One solution would be to enumerate the strings αp in $Pre(M_d)$ with $\alpha \in \Sigma^*$, and then, for each αp enumerate the strings $\alpha p \beta$ in M_d with $\beta \in \Sigma^*$. The resulting enumeration, however,

contains duplicates if there is some string in M_d containing p more than once. Consider the string abaaba which contains $p = \mathtt{ab}$ twice in Example 10. The strings ab ($\alpha = \varepsilon$) and abaab ($\alpha = \mathtt{aba}$) appear in the enumeration of αp, and therefore the string abaaba appears twice in the enumeration of $\alpha p \beta$.

▶ **Example 10.** Let $D = \{\mathtt{aaabaabaaa}, \mathtt{aaabaabbba}, \mathtt{aababababbaa}, \mathtt{abaababbba}\}, d = 2$ and $p = \mathtt{ab}$. Then M_2 is $\{\mathtt{aaabaab}, \mathtt{aabab}, \mathtt{abaaba}, \mathtt{ababb}, \mathtt{abbba}\}$ and the answer is $\{\mathtt{aaabaab}, \mathtt{aabab}, \mathtt{abaaba}, \mathtt{ababb}, \mathtt{abbba}\}$. (1) $\Sigma^* p \cap Pre(M_d) = \{\mathtt{aaab}, \mathtt{aaabaab}, \mathtt{aab}, \mathtt{aabab}, \mathtt{ab}, \mathtt{abaab}, \mathtt{abab}\}$. (2) Their d-left-right-maximal extensions are $\{\mathtt{aaabaab}\}$, $\{\mathtt{aaabaab}\}$, $\{\mathtt{aabab}\}$, $\{\mathtt{aabab}\}$, $\{\mathtt{abaaba}, \mathtt{ababb}, \mathtt{abbba}\}$, $\{\mathtt{abaaba}\}$, $\{\mathtt{ababb}\}$, respectively. (3) The union of these string sets is $\{\mathtt{aaabaab}, \mathtt{aabab}, \mathtt{abaaba}, \mathtt{ababb}, \mathtt{abbba}\}$, which coincides with the answer.

In order to avoid such duplicates in enumeration, we put a restriction on the enumeration of the strings $\alpha p \in Pre(M_d)$. That is, we enumerate the strings $\alpha p \in Pre(M_d)$ satisfying the condition that αp contains p just once, which can be replaced with $LRS(\alpha p) < |p|$. The outline of our algorithm is as follows:

Step 1. Enumerate the strings αp such that $\alpha \in \Sigma^*$, $\alpha p \in Pre(M_d)$ and $LRS(\alpha p) < |p|$.

Step 2. For each string αp obtained in Step 1, enumerate the strings $\alpha p \beta$ such that $\beta \in \Sigma^*$ and $\alpha p \beta \in M_d$.

▶ **Example 11.** Let $D = \{\mathtt{aaabaabaaa}, \mathtt{aaabaabbba}, \mathtt{aababababbaa}, \mathtt{abaababbba}\}, d = 2$ and $p = \mathtt{ab}$. Then M_2 is $\{\mathtt{aaabaab}, \mathtt{aabab}, \mathtt{abaaba}, \mathtt{ababb}, \mathtt{abbba}\}$ and the answer is $\{\mathtt{aaabaab}, \mathtt{aabab}, \mathtt{abaaba}, \mathtt{ababb}, \mathtt{abbba}\}$. (1) $\Sigma^* p \cap Pre(M_d) = \{\mathtt{aaab}, \mathtt{aaabaab}, \mathtt{aab}, \mathtt{aabab}, \mathtt{ab}, \mathtt{abaab}, \mathtt{abab}\}$. (2) Of the seven strings, the three strings aaab, aab, ab satisfy the condition $LRS(x) < |p|$. Their d-right extensions are $\{\mathtt{aaabaab}\}$, $\{\mathtt{aabab}\}$, $\{\mathtt{abaaba}, \mathtt{ababb}, \mathtt{abbba}\}$, respectively. These sets are mutually disjoint. (3) The union of the disjoint sets is $\{\mathtt{aaabaab}, \mathtt{aabab}, \mathtt{abaaba}, \mathtt{ababb}, \mathtt{abbba}\}$, which coincides with the answer (see Figure 1).

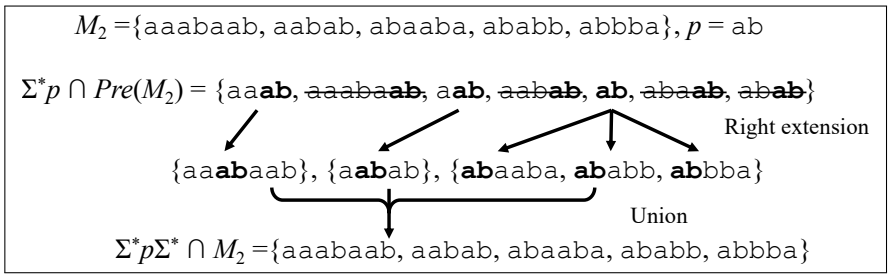

Figure 1 Illustration of Example 11.

4 Simplified Solution

For the sake of simplicity in presentation, we here present a simplified version of our algorithm using an $O(nm)$ space data structure which answers queries in $O(|p| + o)$ time, where o is the number of outputs. How to improve the data structure will be described in the next section.

Basically, we represent substrings of M_d as their loci in $ST(M_d)$. We note that although the strings αp in Step 1 may be represented as implicit nodes of $ST(M_d)$, using their loci does not affect the result of Step 2. The algorithm outline can then be rewritten as follows.

Step 1. Enumerate the loci v of αp in $ST(M_d)$ such that $\alpha \in \Sigma^*$, $\alpha p \in Pre(M_d)$ and $LRS(\alpha p) < |p|$.

Step 2. For each locus v obtained in Step 1, enumerate the loci of $x\beta$ in $ST(M_d)$ such that $\beta \in \Sigma^*$ and $x\beta \in M_d$, where x is the string represented by v in $ST(M_d)$.

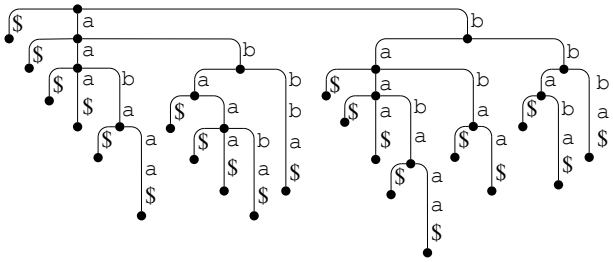

Figure 2 Illustration of Suffix tree $ST({M_2}^R\$)$.

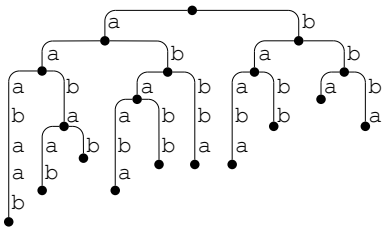

Figure 3 Illustration of Suffix tree $ST(M_2)$.

4.1 Implementation of Step 1

We use the suffix trees $ST({M_d}^R\$)$ for $d = 1, \ldots, m$. We note that there is a natural one-to-one correspondence between the strings x in $Pre(M_d)$ and the leaves of $ST({M_d}^R\$)$ representing $x^R\$$. We also note that for any $p \in \Sigma^+$, the strings in $Pre(M_d) \cap \Sigma^* p$ correspond to the leaves of the subtree rooted at the locus v of p^R in $ST({M_d}^R\$)$. Of the leaves representing $x^R\$$, we have to select those satisfying $LRS(x) < |p|$.

In the running example, the leaves of the subtree rooted at the locus of $p^R = \texttt{ba}$ in $ST({M_2}^R\$)$ represent the strings ba\$, baa\$, baaa\$, baaba\$, baabaaaa\$, baba\$, babaa\$ (see Figure 2). Of the seven strings of the form $x^R\$$, the three strings ba\$, baa\$, baaa\$ satisfy the condition $LRS(x) < |p|$.

Define the function val from the nodes of $ST({M_d}^R\$)$ to the integers by: For any node u of $ST({M_d}^R\$)$, let $val(u) = LRS(x)$ if u is a leaf of $ST({M_d}^R\$)$, and ∞ otherwise, where x is the string such that u represents x^R. By applying the FLT query technique, mentioned in Section 2.4, to the tree $ST({M_d}^R\$)$ with val, we can compute the leaves of $ST({M_d}^R\$)$ representing $(\alpha p)^R\$$ such that $\alpha \in \Sigma^*$, $\alpha p \in Pre(M_d)$ and $LRS(\alpha p) < |p|$. From such a leaf, we can obtain the locus of αp in $ST(M_d)$ in constant time by keeping pointers from the nodes u of $ST({M_d}^R\$)$ to the loci of x in $ST(M_d)$, where x is the string such that x^R is represented by u in $ST({M_d}^R\$)$.

4.2 Implementation of Step 2

We note that the locus of any string in M_d is a leaf of $ST(M_d)$ (see Figure 3). The outputs of Step 2 are thus the leaves u of the subtree rooted at v representing strings in M_d. Define the function val from the nodes of $ST(M_d)$ to the integers by: For any node u of $ST(M_d)$, let $val(u) = 0$ if u is a leaf and represents some string in M_d, and 1 otherwise. We again apply the FLT query technique to the tree $ST(M_d)$ with val, to enumerate the loci of $x\beta$ appropriately.

4.3 Query time and space requirement

In Step 1, computing the locus of p^R in $ST({M_d}^R\$)$ takes $O(|p|)$ time. Each execution of the FLT query takes constant time in Steps 1 and 2. Thus the query time is $O(|p| + o)$, where o is the number of outputs. For $d = 1, \ldots, m$, the suffix trees $ST(M_d)$ and $ST({M_d}^R\$)$, and the relevant data structures for the FLT queries require $O(n)$ space. The total space of our data structure is $O(nm)$.

5 Space Efficient Implementation of Step 1

As seen in Section 4.3, the use of the suffix trees $ST({M_d}^R\$)$ for $d = 1, \ldots, m$ in Step 1 causes the $O(nm)$ space requirement. Our idea to reduce the space requirement is to substitute $ST({M_d}^R)$ for $ST({M_d}^R\$)$. The following lemma gives an upper bound on the total size of suffix trees $ST({M_d}^R)$.

▶ **Lemma 12.** *The suffix trees $ST(M_d)$ for $d = 1, \ldots, m$ are, respectively, of size $O(n/d)$, and their total size is $O(n \log m)$.*

Proof. It suffices to show that $ST(M_d)$ has $O(n/d)$ leaves. Let v be any leaf of $ST(M_d)$, and let x be the string represented by v. Assume, for a contradiction, that x is not d-right maximal. Then, there exists a string $\beta \in \Sigma^+$ such that $\alpha x \beta \in M_d$ for some $\alpha \in \Sigma^*$. Thus $x\beta$ is a suffix of M_d, which contradicts that v is a leaf of $ST(M_d)$. Therefore x is d-right maximal. The number of leaves of $ST(M_d)$ is not greater than the number of d-right maximal strings, which is $O(n/d)$ by Lemma 2. ◀

The difficulty in using not $ST({M_d}^R\$)$ but $ST({M_d}^R)$ is that the string $(\alpha p)^R$ is possibly implicit in $ST({M_d}^R)$ whereas the string $(\alpha p)^R\$$ is necessarily explicit and represented by a leaf in $ST({M_d}^R\$)$. We partition Step 1 into two parts:

Step 1A. Enumerate the loci of αp in $ST(M_d)$ such that $\alpha \in \Sigma^*$, $\alpha p \in Pre(M_d)$, $LRS(\alpha p) < |p|$ and $(\alpha p)^R$ is explicit in $ST({M_d}^R)$.

Step 1B. Enumerate the loci of αp in $ST(M_d)$ such that $\alpha \in \Sigma^*$, $\alpha p \in Pre(M_d)$, $LRS(\alpha p) < |p|$ and $(\alpha p)^R$ is implicit in $ST({M_d}^R)$.

Step 1A can be done in $O(|p| + o)$ time with $O(n \log m)$ space in almost the same way as Section 4.1. Below we describe how to implement Step 1B.

5.1 Implementation of Step 1B

▶ **Lemma 13.** *For any string x in $Pre(M_d)$, x^R is explicit in $ST(D^R\$)$.*

Proof. Let $\beta \in \Sigma^*$ be a string such that $x\beta \in M_d$. Since the string $(x\beta)^R$ is d-left-right maximal, it is explicit in $ST(D^R\$)$ and therefore its suffix x^R is also explicit in $ST(D^R\$)$. ◀

We thus use $ST(D^R\$)$ to represent strings in $Pre(M_d)$.

Let q_1 and q_2 be the strings represented by the loci of p^R in $ST(D^R\$)$ and in $ST({M_d}^R)$, respectively. The *p-critical* path of $ST(D^R\$)$ is the path from u_1 to u_2 such that u_1 and u_2 are the nodes of $ST(D^R\$)$ representing q_1 and q_2, respectively. A string x and the node representing x^R in $ST(D^R\$)$ are said to be *p-satisfying* if x is a left extension of p such that $x \in Pre(M_d)$, $LRS(x) < |p|$ and x^R is implicit in $ST({M_d}^R)$. An edge e of $ST({M_d}^R)$ and the path corresponding to e in $ST(D^R\$)$ are said to be *p-admissible* if e is in the subtree rooted at the node representing q_2 and at least one implicit node is present on e which represents the reversal x^R of a p-satisfying string x.

Every p-satisfying node of $ST(D^R\$)$ is present on: (i) the p-critical path of $ST(D^R\$)$ or (ii) a p-admissible path of $ST({M_d}^R)$. Thus, the enumeration of the loci of αp in $ST(M_d)$ can be performed as follows.

(1) Enumerate the p-admissible paths of $ST(D^R\$)$.
(2) For each p-admissible path of $ST(D^R\$)$ and for the p-critical path of $ST(D^R\$)$, enumerate the p-satisfying nodes on it.
(3) For each p-satisfying node of $ST(D^R\$)$ representing x^R, compute the locus of x in $ST(M_d)$.

▶ **Example 14.** Suppose that $D = \{\texttt{aaabaabaaa}, \texttt{aaabaabbba}, \texttt{aababababaa}, \texttt{abaababbba}\}$, $d = 2$ and $p = \texttt{abab}$. Then, $q_1 = \texttt{baba}$ and $q_2 = \texttt{babaaa}$ (see Figure 4). We want to compute $\texttt{baba}$ in Step 1B.

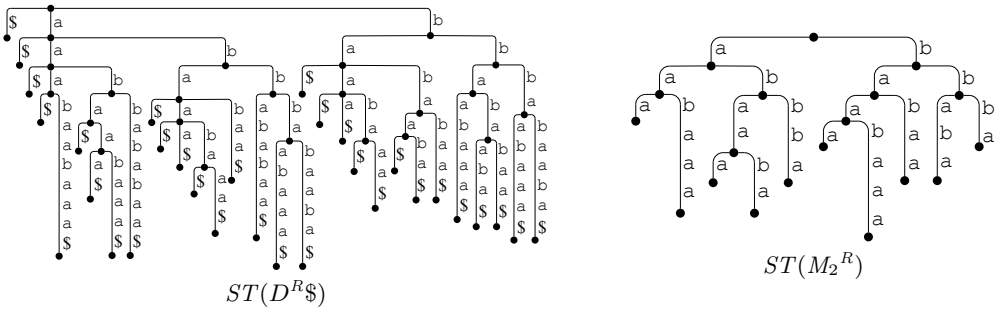

Figure 4 $ST(D^R\$)$ and $ST({M_2}^R)$ for $D = \{\texttt{aaabaabaaa}, \texttt{aaabaabbba}, \texttt{aababababaa}, \texttt{abaababbba}\}$.

In (1), we shall enumerate all p-admissible edges of $ST({M_d}^R)$. With each edge (s, t) of $ST({M_d}^R)$, we associate the value $LRS(z^R)$ such that x and y are the strings represented by s and t, respectively, and $z = y[..i]$ where i is the smallest integer in $[|x| + 1, |y| - 1]$ with $z \in Suf({M_d}^R)$ (i.e. $z^R \in Pre(M_d)$). We associate ∞ with it, if no such i exists. Then, we can enumerate all p-admissible edges of $ST({M_d}^R)$, by applying the FLT query technique to $ST({M_d}^R)$, with regarding the value associated with the incoming edge (s, t) of a node t as the value of t. Computing the loci of p^R in $ST({M_d}^R\$)$ and $ST(D^R\$)$ takes $O(|p|)$ time. Execution of the FLT query takes constant time.

In (2), we proceed to examine nodes representing x^R on the path until we encounter a node representing x^R with $LRS(x) \geq |p|$, by repeatedly querying with the data structure stated in the following lemma.

▶ **Lemma 15.** *There exists an $O(n \log m)$ size data structure which, given a node of $ST(D^R\$)$ representing string y^R, returns in $O(\log \log m)$ time the locus of $(xy)^R$ in $ST(D^R\$)$ such that x is the shortest string with $xy \in Pre(M_d)$, and nil if no such x exists.*

In (3), for each p-satisfying node of $ST(D^R\$)$ representing x^R, compute the locus of x in $ST(M_d)$ by using the data structure stated in the following lemma.

▶ **Lemma 16.** *The locus of a string x in $ST(M_d)$ can be computed in $O(\log \log m)$ time from the locus of x^R in $ST(D^R\$)$ using an $O(n \log m)$ space data structure.*

The suffix tree $ST(D^R\$)$ takes $O(n)$ space. For $d = 1, \ldots, m$, the suffix trees $ST(M_d)$ and $ST({M_d}^R\$)$, and the relevant data structures for the FLT queries require $O(n)$ space. The total computation time of Step 1B is $O(|p| + o \log \log m)$ and the total space of our data structure is $O(n \log m)$.

5.2 Proofs of Lemmas 15 and 16

To complete the proof of Theorem 9, we give proofs of Lemmas 15 and 16. For the sake of convenience, we first prove Lemma 16.

5.2.1 Proof of Lemma 16

From the locus of x^R in $ST(D^R\$)$ we can obtain the locus of x in $ST(D\$)$ in constant time by using direct links from the nodes of $ST(D^R\$)$ to the corresponding nodes of $ST(D\$)$. Thus we describe how to compute from the locus of x in $ST(D\$)$ the locus of x in $ST(M_d)$ in $O(\log\log m)$ time using $O(n\log m)$ space.

A node v of $ST(D\$)$ representing string z is called a d-node if z is explicit in $ST(M_d)$. The locus of x in $ST(M_d)$ then corresponds to the earliest d-node preceded by the locus of x in the pre-order traversal of $ST(D\$)$.

▶ **Example 17.** Suppose that $D = \{\texttt{aaabaabaaa}, \texttt{aaabaabbba}, \texttt{aababababbaa}, \texttt{abaababbba}\}$, $d = 2$ and $x = \texttt{aab}$. Then the earliest 2-node preceded by the locus of x is the node representing `aaba` (see Figure 5). The locus of x in $ST(M_2)$ represents the same string `aaba`.

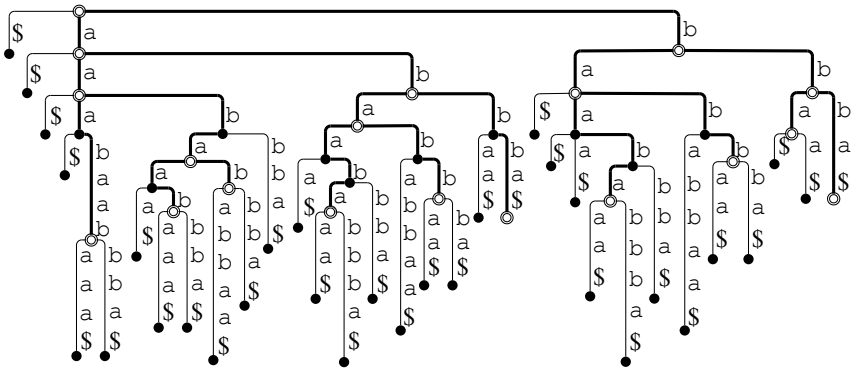

Figure 5 Illustration of $ST(D\$)$, where the double lined circles represent the 2-nodes.

For any node s of $ST(D\$)$, let $L(s)$ be the sequence of non-negative integers d arranged in the increasing order such that $d = 0$ or s is a d-node. Let A be the sequence obtained by concatenating $L(s)$ according to the pre-order of nodes s of $ST(D\$)$. Let u and v be the loci of x in $ST(D\$)$ and $ST(M_d)$, respectively. Then v corresponds to the leftmost occurrence of d in $A[i+1..]$ such that i is the position of j-th occurrence of 0 where j is the rank of u in the pre-order traversal of $ST(D\$)$. Thus v can be computed from u as follows. For the rightmost leaf l_u of the subtree rooted at u, $v = nil$ if $rank_A(d, select_A(0, PreOrd(u))) > select_A(0, PreOrd(l_u))$, and otherwise, v corresponds to $A[rank_A(d, select_A(0, PreOrd(u)))]$, where $PreOrd(s)$ denotes the rank of a node s in the pre-order traversal of $ST(D\$)$.

The numbers of 0's and d's in the array A are $O(n)$ and $O(n/d)$, respectively, and hence we have $|A| = O(n\log m)$. By Lemma 5, we can compute the locus of x in $ST(M_d)$ from the locus of x in $ST(D^R\$)$ in $O(\log\log m)$ time using $O(n\log m)$ space.

5.2.2 Proof of Lemma 15

By Lemma 3, $xyz \in M_d$ implies that yz is d-right maximal. For each d-right maximal string β, let $len(\beta)$ denote the length of the shortest string α with $\alpha\beta \in M_d$. Then the desired string xy can be obtained from the d-right maximal extension yz of y that minimizes $len(yz)$.

From the locus of y^R in $ST(D^R\$)$, the locus of $(xy)^R$ in $ST(D^R\$)$ can be computed in three steps (see Figure 6).

Step 1. From the locus of y^R in $ST(D^R\$)$, find the locus of y in $ST(M_d)$.

Step 2. From the locus of y in $ST(M_d)$, find the locus of $(xyz)^R$ in $ST(D^R\$)$ such that x is one of the shortest strings x satisfying $xyz \in M_d$ for some string z.

Step 3. From the locus of $(xyz)^R$, find the locus of $(xy)^R$ in $ST(D^R\$)$.

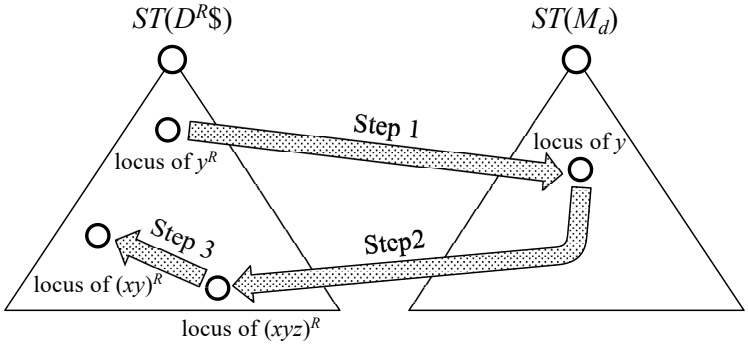

Figure 6 Computing the locus of $(xy)^R$ from the locus of y^R in $ST(D^R\$)$.

Step 1 requires $O(\log \log m)$ time by using the $O(n \log m)$-size data structure stated in Lemma 16.

For Step 2, we define two functions *len* and *link* on the set of nodes of $ST(M_d)$ as follows: For any node u of $ST(M_d)$, let β be the string represented by u. If there is some string α such that $\alpha\beta \in Pre(M_d)$, choose α as short as possible, and let $len(u) = |\alpha|$ and let $link(u)$ be the locus of α in $ST(D^R\$)$. If there is no such α, let $len(u) = \infty$ and $link(u) = nil$.

Suppose that v is the descendant of the locus of y in $ST(M_d)$ that minimizes $len(v)$. Then $len(v) = |x|$ and $link(v)$ is the locus of $(xyz)^R$ in $ST(D^R\$)$ since yz is d-right maximal. The locus of $(xyz)^R$ in $ST(D^R)$ can then be computed in constant time by storing the values $len(u)$ and $link(u)$ into the nodes u of $ST(M_d)$ and applying the Range Minimum Query technique.

In Step 3, the locus of $(xy)^R$ is obtained from the locus of $(xyz)^R$ in $ST(D^R\$)$ by traversing suffix links $|x|$ times. The task can be done in constant time by using the $O(n)$ space data structure for the level ancestor query [1] on suffix link tree of $ST(D^R\$)$.

Step 1 through Step 3 can be done in $O(\log \log m)$ time using $O(n \log m)$ space.

6 Conclusion

In this paper, we addressed the left-right maximal generic words problem and developed an $O(n \log m)$ size data structure, which answers queries in $O(|p| + o \log \log m)$ time, where o is the size of outputs. Our method is better than the previous work by Nishimoto et al. [7] both in the space requirement and in the query time. We achieved the $O(n \log m)$ space requirement by substituting $ST(M_d)$ for $ST(M_d\$)$, with the conjecture that the total size of $ST(M_d\$)$'s for $d = 1, \ldots, m$ are $\Theta(nm)$. To prove that the total size is $\Omega(nm)$ is left as future work.

References

1 Omer Berkman and Uzi Vishkin. Finding Level-Ancestors in Trees. *J. Comput. Syst. Sci.*, 48(2):214–230, 1994.

2 Sudip Biswas, Manish Patil, Rahul Shah, and Sharma V. Thankachan. Succinct Indexes for Reporting Discriminating and Generic Words. In Edleno Silva de Moura and Maxime Crochemore, editors, *String Processing and Information Retrieval - 21st International Symposium, SPIRE 2014, Ouro Preto, Brazil, October 20-22, 2014. Proceedings*, volume 8799 of *Lecture Notes in Computer Science*, pages 89–100. Springer, 2014. doi:10.1007/978-3-319-11918-2.

3 Pawel Gawrychowski, Gregory Kucherov, Yakov Nekrich, and Tatiana A. Starikovskaya. Minimal Discriminating Words Problem Revisited. In Oren Kurland, Moshe Lewenstein, and Ely Porat, editors, *String Processing and Information Retrieval - 20th International Symposium, SPIRE 2013, Jerusalem, Israel, October 7-9, 2013, Proceedings*, volume 8214 of *Lecture Notes in Computer Science*, pages 129–140. Springer, 2013. doi:10.1007/978-3-319-02432-5.

4 Alexander Golynski, J. Ian Munro, and S. Srinivasa Rao. Rank/select operations on large alphabets: a tool for text indexing. In *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2006, Miami, Florida, USA, January 22-26, 2006*, pages 368–373. ACM Press, 2006. URL: http://dl.acm.org/citation.cfm?id=1109557.1109599.

5 Gregory Kucherov, Yakov Nekrich, and Tatiana A. Starikovskaya. Computing Discriminating and Generic Words. In Liliana Calderón-Benavides, Cristina N. González-Caro, Edgar Chávez, and Nivio Ziviani, editors, *String Processing and Information Retrieval - 19th International Symposium, SPIRE 2012, Cartagena de Indias, Colombia, October 21-25, 2012. Proceedings*, volume 7608 of *Lecture Notes in Computer Science*, pages 307–317. Springer, 2012. doi:10.1007/978-3-642-34109-0.

6 S. Muthukrishnan. Efficient algorithms for document retrieval problems. In David Eppstein, editor, *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms, January 6-8, 2002, San Francisco, CA, USA.*, pages 657–666. ACM/SIAM, 2002. URL: http://dl.acm.org/citation.cfm?id=545381.545469.

7 Takaaki Nishimoto, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Computing Left-Right Maximal Generic Words. In Jan Holub and Jan Zdárek, editors, *Proceedings of the Prague Stringology Conference 2015, Prague, Czech Republic, August 24-26, 2015*, pages 5–16. Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague, 2015. URL: http://www.stringology.org/event/2015/p02.html.

8 Peter Weiner. Linear Pattern Matching Algorithms. In *14th Annual Symposium on Switching and Automata Theory, Iowa City, Iowa, USA, October 15-17, 1973*, pages 1–11, 1973.