Computing Left-Right Maximal Generic Words

Takaaki Nishimoto[1], Yuto Nakashima[1], Shunsuke Inenaga[1], Hideo Bannai[1], and Masayuki Takeda[1]

Department of Informatics, Kyushu University, Japan
{takaaki.nishimoto, yuto.nakashima, inenaga, bannai, takeda}@inf.kyushu-u.ac.jp

Abstract. The *maximal generic words problem* was proposed by Kucherov et al. (SPIRE 2012). Let D be a set of documents. In this problem, given a pattern P and a threshold $d \leq |D|$, we want to compute all *right-maximal* extensions of P which occur in at least d distinct documents. They proposed an $O(n)$-space data structure which can solve this problem in $O(|P| + rocc)$ time where n is the total length of documents in D and $rocc$ is the number of right-maximal extensions of P. The data structure can be constructed in $O(n)$ time. In this paper, we propose a more generalized problem. Given a pattern P and a threshold $d \leq |D|$, we want to compute all *left-right-maximal* extensions of P which occur in at least d distinct documents. We propose an $O(n \log n)$-space data structure which can solve this problem in $O(|P| + occ \log^2 n + rocc \log n)$ time where occ is the number of left-right-maximal extensions of P.

1 Introduction

Let $D = \{T_1, \ldots, T_m\}$ be a set of strings of total n characters from an alphabet Σ, called *documents*. Kucherov et al. [8] proposed the *right-maximal generic words problem*: Given a pattern P and threshold $d \leq m$, return all maximal right extensions of P which occur in at least d distinct documents, where a right extension of P is a string which has P as a prefix. This problem is important to applications in computational biology, text mining, and text classification (see [8,3,7]). Kucherov et al. [8] solved the right-maximal generic words problem in $O(|P| + rocc)$ query time, $O(n)$ preprocessing time, and using $O(n)$ space, where $rocc$ is the number of the output right-maximal extensions. Later, Biswas et al. [3] proposed a succinct data structure of $n \log |\Sigma| + o(n \log |\Sigma|) + O(n)$ bits of space which solves the right-maximal generic words problem in $O(|P| + \log \log n + rocc)$ query time.

In this paper, we consider a more generalized problem: Given a pattern P and threshold $d \leq m$, return all maximal left-right extensions of P which occur in at least d documents, where a left-right extension of P is a superstring of P. For example, let $D = \{T_1, \ldots, T_4\}$, where $T_1 = bababaa\$_1$, $T_2 = ccabacc\$_2$, $T_3 = abaaccabab\$_3$, and $T_4 = cabacbaba\$_4$. Given a pattern $P = aba$ and threshold $d = 2$, then the answer is $\{cabac, abaa, ccaba, baba, abab\}$.

Since all right-maximal generic words of a given pattern P have P as a prefix, the right-maximal generic words problem can be solved by using the generalized suffix tree of D and segment intersection query data structure [4], in linear total space. In contrast, left-right-maximal generic words of P are superstrings of P, and hence P is not necessarily a prefix of the solutions. If we construct the generalized suffix trees of all substrings of documents in D and segment intersection query data structures, we would be able to quickly answer the left-right-maximal extensions of P. However, obviously this approach requires $\Omega(n^2)$ space. Hence, our left-right-maximal generic words problem seems more complicated than the right-maximal generic words problem.

Takaaki Nishimoto, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, Masayuki Takeda: Computing Left-Right Maximal Generic Words, pp. 5–16.
Proceedings of PSC 2015, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-05787-2 © Czech Technical University in Prague, Czech Republic

In this paper, we propose an $O(n \log n)$-space solution to the left-right-maximal words problem using the following data structures:

(i) A data structure which, given a string P and threshold d, finds all right extensions $x_1, \ldots, x_k$ of P satisfying the following properties: for $1 \leq i \leq k$,
   (1) x_i contains P only as a prefix.
   (2) There exists at least one left-right-maximal extension of x_i which occurs in at least d documents and contains x_i as a suffix. Note that every left-right-maximal extension of x_i is left-right-maximal extension of P.
   (3) x_i occurs in at least d distinct documents.
   The running time is $O(k \log^2 n + \mathit{rocc} \log n)$, where *rocc* is the number of answers of the right-maximal generic words problem with a given pattern P and threshold d.

(ii) A data structure which, given a string P, finds all left-right-maximal extensions of P which occur in at least d documents and contains P as a suffix in $O(\log \log n + c)$ time, where c is the number of the outputs.

Our algorithm is summarized as follows: (a) Firstly, we compute all right extensions $x_1, \ldots, x_k$ of P satisfying (1)(2)(3) using (i). (b) Secondly, for $1 \leq i \leq k$, we compute all maximal left-right extensions of P which occur in at least d distinct documents and contains x_i as a suffix using (ii). Hence we can solve our problem in $O(|P| + \mathit{occ} \log^2 n + \mathit{rocc} \log n)$ time.

2 Preliminaries

2.1 Strings

Let Σ be a finite *alphabet*. An element of Σ^* is called a *string*. The length of a string w is denoted by $|w|$. The empty string ε is a string of length 0, namely, $|\varepsilon| = 0$. For a string $w = xyz$, x, y and z are called a *prefix*, *substring*, and *suffix* of w, respectively. A prefix x and a suffix z of w are called a *proper prefix* and a *proper suffix* of w if $x \neq w$ and $z \neq w$, i.e., x and y is shorter than w, respectively. The i-th character of a string w is denoted by $w[i]$, where $1 \leq i \leq |w|$. For a string w and two integers $1 \leq i \leq j \leq |w|$, let $w[i..j]$ denote the substring of w that begins at position i and ends at position j. For convenience, let $w[i..j] = \varepsilon$ when $i > j$. For any string w, let w^R denote the reversed string of w, i.e., $w^R = w[|w|]w[|w| - 1] \cdots w[1]$.

Let $D = \{T_1, \ldots, T_m\}$ be a set of strings. The set of reversed strings of D is denoted by $D' = \{T_1^R, \ldots, T_m^R\}$. For any strings x and y, let $x \cdot y$ denote the concatenation of x and y.

A *right extension* of a string w is a string which has w as a prefix, a *left extension* of a string w is a string which has w as a suffix, and a *left-right extension* of a string w is a string which has w as a substring.

2.2 *d*-maximal

For any set D of documents (strings) and any string w, let $\mathit{Weight}_D(w)$ denote the number of distinct documents in D which have w as a substring. A string x is said to be *d-right-maximal* if $\mathit{Weight}_D(x) \geq d$ and $\mathit{Weight}_D(xa) < d$ for any $a \in \Sigma$. A string x is said to be *d-left-maximal* if $\mathit{Weight}_D(x) \geq d$ and $\mathit{Weight}_D(ax) < d$ for any $a \in \Sigma$. A string x is said to be *d-left-right-maximal* if x is *d-right-maximal* and *d-left-maximal*. Throughout the paper, the total length of documents in D will be denoted by n.

2.3 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).

2.4 Tools

Generalized Suffix Trees. Let $\mathcal{T}$ be any edge labeled tree. For any node u of $\mathcal{T}$, let $str_{\mathcal{T}}(u)$ denote the string which is a concatenation of the edge labels from the root to u. We will abbreviate $str_{\mathcal{T}}(u)$ as $str(u)$ when clear from the context.

A *generalized suffix tree* of a set of strings is the suffix tree [11] that contains all suffixes of all the strings in the set. We denote a *generalized suffix tree* of D by GST_D. We define some notations and additional information of GST_D and $GST_{D'}$.

Let V_{GST_D} be the set of nodes of GST_D. The subtree rooted at node u is denoted by $GST_D(u)$. For any string x, let $L(x)$ be the node u which is the highest node in V_{GST_D} s.t. x is a prefix of $str(u)$. We denote the locus of x in $GST_{D'}$ by $L'(x)$. As in the previous work [8,7], we assume that $L(x)$ and $L'(x)$ for a given string x can be computed in $O(|x|)$ time using GST_D and $GST_{D'}$, respectively[1]. Let $weight(u)$ be the number of distinct documents in D which have $str(u)$ as a substring, and let $maxchild(u) = \max\{weight(v) \mid v \text{ is a child of } u\}$.

Tools on Trees. For any nodes $u, v \in V_{GST_D}$, let $LCA(u, v)$ denote the lowest common ancestor (LCA) of u and v. We can preprocess the tree in linear time so that for any nodes u, v, $LCA(u, v)$ can be computed in constant time (e.g. [1]). For any node $u \in V_{GST_D}$ and integer $d \geq 0$, let $LA(u, d)$ denote the depth-d ancestor (*level ancestor*) of node u in GST_D. We can preprocess the tree in linear time so that for any node u and integer d, we can compute $LA(u, d)$, in constant time (e.g. [2]).

Segment Intersection Query. A horizontal segment $([x, x'], y)$ on a 2D plane (resp. vertical segment $(x, [y, y'])$) is a line connecting points (x, y) and (x', y) (resp. points (x, y) and (x, y')). We say that a vertical segment $p = (x_p, [y_p, y'_p])$ *stabs* a horizontal segment $q = ([x_q, x'_q], y_q)$ if $x_q \leq x_p \leq x'_q$ and $y_p \leq y_q \leq y'_p$. *Segment Intersection Queries* for a horizontal segment set $\mathcal{S}$ are: given a vertical segment p, return the subset of segments of $\mathcal{S}$ that p stabs. There exist many data structures for this segment intersection queries [5,6,4]. In this paper, we use the data structure that occupies $O(|\mathcal{S}|)$ space and supports segment intersection queries in $O(\log\log|\mathcal{S}| + k)$ time, where k is the output size [4]. Next, suppose that each segment in $q \in \mathcal{S}$ is associated with an integer weight $w(q) \geq 0$. *Segment Intersection Sum Queries* are: given a vertical segment p, return the total sum of weights of the segments in $\mathcal{S}$ that p stabs. Since segment intersection sum queries are a special case of *rectangle intersection sum queries*, there exists a data structure occupies $O(|\mathcal{S}| \max\{1, \log \frac{W}{|\mathcal{S}|}\})$ space and supports segment intersection sum queries in $O(\log|\mathcal{S}|)$ time, where $W = \sum_{q \in \mathcal{S}} w(q)$ [10]. Similarly, we define *Segment Intersection Count Queries* for a horizontal segment set $\mathcal{S}$. This is a special case of the segment intersection sum query where $w(q) = 1$ for every segment $q \in \mathcal{S}$. Hence there exists a data structure occupies that $O(|\mathcal{S}|)$ space and supports segment intersection count queries in $O(\log|\mathcal{S}|)$ time.

[1] If $|\Sigma|$ is constant, then clearly $L(x)$ and $L'(x)$ can be computed in $O(|x|)$ time. If $|\Sigma|$ is not constant, these can be computed in expected $O(|x|)$ time using hashing.

3 Problem and Properties

In this paper, we consider the following problem.

Problem 1. Let $D = \{T_1, \ldots, T_m\}$ be a set of documents. Given a pattern P and positive integer d ($\leq m$), compute all d-left-right-maximal extensions of P.

Let $Ans_D(P, d)$ be the set of answers to Problem 1. In Section 3.1, we show a relation between $Ans_D(P, d)$ and V_{GST_D}. In this paper, we output answers as a set of nodes in $GST_{D'}$.

3.1 Relation between Answers and GST

Lemma 2. *For any $z \in Ans_D(P, d)$, there exists a node $u \in V_{GST_D}$ s.t. $str(u) = z$.*

Proof. Since z is a substring of some document, we can traverse GST_D from the root with z. Because $z \in Ans_D(P, d)$, the number of occurrences of z and za in D are different for any $a \in \Sigma$. Thus there exists a node u s.t. $str(u) = z$. □

The following corollary can be easily obtained in a similar way.

Corollary 3. *For any $z \in Ans_D(P, d)$, there exists a node $u' \in V_{GST_{D'}}$ s.t. $str(u')^R = z$.*

Let Occ be the set of nodes $u' \in V_{GST_{D'}}$ s.t. $str(u')^R \in Ans_D(P, d)$. It follows from the above corollary that Occ is the set of nodes in $V_{GST_{D'}}$ that represent all occ answers to Problem 1. In this paper, we will compute Occ as an output.

Our main idea is the following. First, we choose a node $u \in V_{GST_D(L(P))}$. This node represents a (not necessarily maximal) *right extension* of P. Second, we compute the nodes $u' \in V_{GST_{D'}(L'(str(u)^R))}$ s.t. $u' \in Occ$. By the condition $u' \in Occ$, it is clear that $weight(u') \geq d$ and $maxchild(u') < d$ holds, and hence these nodes represent d-left-right-maximal extensions of P (see also Fig. 1). Thus, if we conduct the above procedure for all nodes in $u \in V_{GST_D(L(P))}$, we can obtain all solutions to Problem 1.

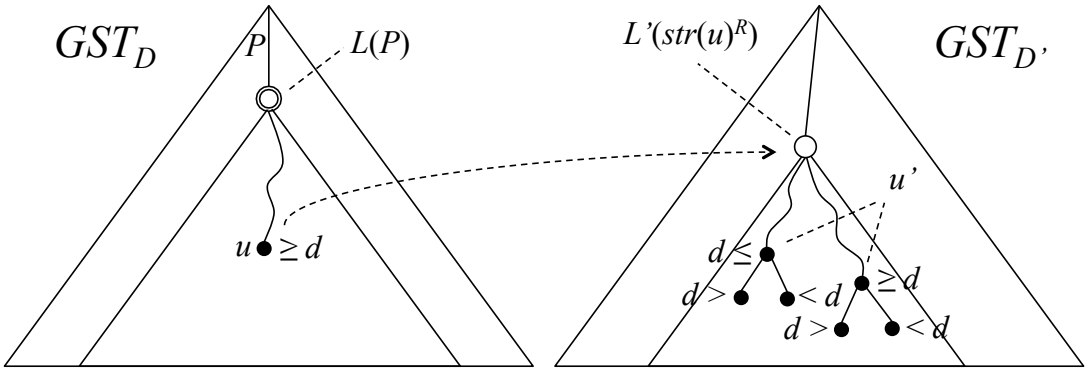

Figure 1. This is a conceptual diagram of our main idea.

In what follows, Section 3.2 characterizes the set of nodes in $GST_{D'}$ which represent (a subset of) the answers to the problem w.r.t. a given *right extension* of query pattern P. Note that the strings represented by these nodes have the *right extension* of P as a suffix. Then, Section 3.3 characterizes a subset of *right extensions* of P which are sufficient to compute all d-left-right-maximal extensions of P without duplicates.

3.2 *d*-Left-Right-Maximal Extensions of a Given Right Extension

First, we define a set $cand(u)$ for any node $u \in V_{GST_D}$ that represents *left extensions* of $str(u)$ which are d-left-maximal. Let $\ell'_u = L'(str(u)^R) \in V_{GST_{D'}}$, and

$$cand(u) = \{u' \mid u' \in V_{GST_{D'}(\ell'_u)}, weight(u') \geq d \text{ and } maxchild(u') < d\}.$$

We also define $Cand(V) = \cup_{u \in V} cand(u)$ for each $V \subseteq V_{GST_D}$. In our algorithm, given a *right extension* x of P that occurs in at least d distinct documents, we compute d-left-maximal extensions of x. Let $REx = \{u \mid u \in V_{GST_D(L(P))} \text{ and } weight(u) \geq d\}$. Figure 2 gives examples of some definitions.

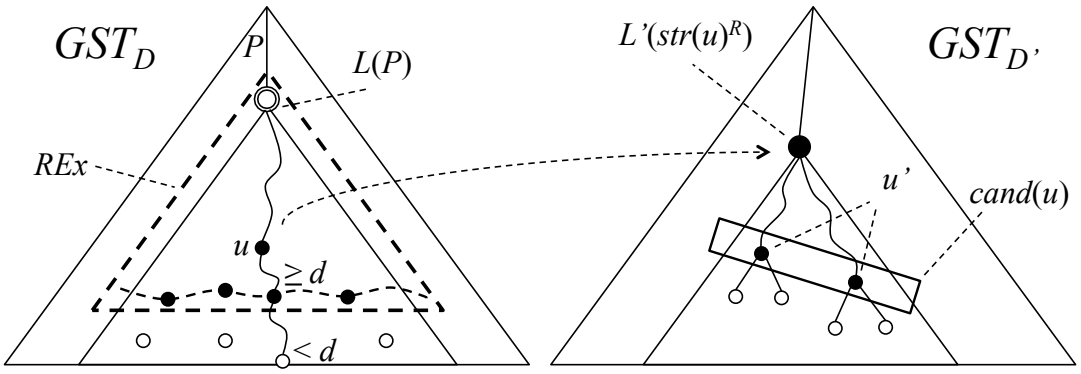

Figure 2. This figure shows examples of $cand(u)$ and REx. The black circles represent nodes s.t. its weight is larger than d. The white circles represent nodes s.t. its weight is strictly smallar than d.

Lemma 4. *$Cand(REx) \supseteq Occ$.*

Proof. Let u' be a node in Occ and let $z = str(u')^R \in Ans_D(P, d)$. Since $z \in Ans_D(P, d)$, $weight(u') \geq d$ and $maxchild(u') < d$ hold. On the other hand, there exists a node $w \in GST_D$ s.t. $z = str(w)$ by Lemma 2. Let z_1 be a suffix of z which has P as a prefix. Then there is a node $u \in GST_D$ s.t. $z_1 = str(u)$. It also holds that $weight(u) \geq d$, and z_1 has P as a prefix, so $u \in REx$. Since $str(u)^R$ is a prefix of z^R, $u' \in cand(u)$. Thus $u' \in Cand(REx)$. Therefore this lemma holds. □

Any string which is represented by a node in $Cand(REx)$ is guaranteed only to be d-left-maximal in D. Thus, there may exist a node in $Cand(REx) \cap \overline{Occ}$, where $\overline{Occ} = V_{GST_{D'}} - Occ$. To remove such nodes from $Cand(REx)$, the following lemma characterizes such nodes.

Lemma 5. *For any $u' \in Cand(REx) \cap \overline{Occ}$, there exists a node $v' \in Occ$ s.t. $str(v')^R$ has $str(u')^R$ as a proper prefix.*

Proof. By the definition of $Cand(REx)$, it is clear. □

By using the above lemma, we want to remove the nodes in $Cand(REx) \cap \overline{Occ}$ from $Cand(REx)$. For any node $u' \in V_{GST_{D'}}$ and any character c, let $FC(u', c)$ be the number of distinct documents in D which have $str(u')^R \cdot c$ as a substring. We define $MFC(u') = \max\{FC(u', c) \mid \forall c \in \Sigma\}$. $MFC(u')$ represents the maximum number of strings in D which have $str(u')^R \cdot c$ as a substring for any character c. We use $MFC(u')$ to remove nodes which are not in Occ from $Cand(REx)$.

Lemma 6. *For any node $u' \in cand(u)$ for some $u \in REx$, $MFC(u') \geq d$ iff $u' \notin Occ$.*

Proof. ($\Rightarrow$). By the definition of $MFC(u')$, $Weight_D(str(u')^R \cdot c) \geq d$ for some $c \in \Sigma$ holds. Thus $u' \notin Occ$. ($\Leftarrow$). It is clear from Lemma 5. □

Corollary 7. *For any node $u' \in cand(u_1)$ for some $u_1 \in REx$, If $MFC(u') \geq d$, then there exists a node u_2 which is a descendant of u_1 s.t. $cand(u_2) \cap Occ \neq \phi$.*

Proof. By Lemmas 5, 6, there exists $z = str(u')^R \cdot x \in Ans_D(P, d)$ for some $x \in \Sigma^+$. From Lemma 2, there is a node $w \in GST_D$ s.t. $str(w) = z$. Thus there is also a node u_2 s.t. $str(u_2) = str(u_1) \cdot x$. It is clear that u_2 is a descendant of u_1. □

Now we define a new set $Cand_1(REx)$ of nodes in $GST_{D'}$ s.t. $Cand'_1(REx) = Occ$. For any $u \in V_{GST_D}$, let

$$cand_1(u) = \{u' \mid u' \in V_{GST_{D'}(\ell'_u)}, weight(u') \geq d, maxchild(u') < d \text{ and } MFC(u') < d\}.$$

We define $Cand_1(V) = \cup_{u \in V} cand_1(u)$ for any $V \subseteq V_{GST_D}$.

Lemma 8. *$Cand_1(REx) = Occ$.*

Proof. It is clear from Lemmas 5, 6. □

3.3 Meaningful Right Extensions of P

Let $REx = \{u_1, \ldots, u_h\}$. By Lemma 8, $Cand_1(REx)$ and Occ are equivalent, but $|cand_1(u_1)| + \cdots + |cand_1(u_h)| \geq |Occ|$ holds. The following lemma characterizes this situation.

Lemma 9. *Let u_1 be a node in REx s.t. P occurs in $str(u_1)$ at least two times. For any node $u_2 \in REx$ s.t. $str(u_2)$ is a proper suffix of $str(u_1)$, $cand_1(u_1) \subseteq cand_2(u_2)$.*

Proof. For any $u' \in cand_1(u_1)$, $str(u_2)$ is a suffix of $str(u')^R$. Thus $str(u')^R \in cand_1(u_2)$. Therefore this lemma holds. □

We define a new set REx_1 s.t. $REx_1 \subseteq REx$. Let $REx_1 = \{u \mid u \in V_{GST(L(P))}$, $weight(u) \geq d$, and P occurs in $str(u)$ only as a prefix $\}$. By the above lemma, the following lemma holds.

Lemma 10. *$Cand_1(REx_1) = Occ$ and $\sum_{u \in REx_1} |cand_1(u)| = |Occ|$ hold.*

Proof. By Lemmas 8, 9, $Cand_1(REx_1) = Occ$ holds. Let u_1 and u_2 be elements of REx_1 s.t. $u_1 \neq u_2$. We assume $|str(u_1)| \leq |str(u_2)|$. By the definition of REx_1, $str(u_1)$ is not a suffix of $str(u_2)$. Thus $cand_1(u_1) \cap cand_1(u_2) = \phi$. Therefore $\sum_{u \in REx_1} |cand_1(u)| = |Occ|$ holds. □

Clearly, there may exist a node $u \in REx_1$ s.t. $cand_1(u) = \phi$. From Lemma 10, we do not want to compute $cand_1(u)$ for such $u \in REx_1$. Let $REx_2 = \{u \mid cand_1(u) \neq \phi\}$. For any $u \in REx_2$, $cand_1(u) \neq \phi$, $\cup_{u \in REx_2} cand_1(u) = Ans_D(P, d)$ and $cand_1(u_1) \cap cand_1(u_2)$ for any u_1 and u_2.

In the rest of this section, we show some lemmas which are useful to compute REx_2 efficiently. For any $u \in REx_1$, let $r'(u)$ be a node in $GST_{D'}$ s.t. $str(r'(u)) = str(u)^R$ ($r'(u)$ may be an *implicit node*). We define T_u as a tree which is a subgraph of $GST_{D'}(r'(u))$ s.t. the root is $r'(u)$ and leaves are all nodes in $cand(u)$. In fact, T_u

represents left extended strings of $str(u)$. Figure 3 shows an example of T_u. Let $Leaf(T_u)$ be a set of all leaves in T_u, and $size(T_u) = \sum_{v \in Leaf(T_u)} |str(v)|$. Then $size(T_{u_1}) - size(T_{u_2}) \geq 0$ for any u_1 and u_2 in REx_1 s.t. u_2 is a child of u_1 holds. To prove this, we show T_{u_2} can be superimposed on T_{u_1} for any u_1 and u_2 in REx_1 s.t. u_2 is a descendant of u_1. If there exists a node v in T_{u_1} s.t. $str(w)$ is a prefix of $str(v)$ for each leaf w in T_{u_2}, T_{u_2} can be superimposed on T_{u_1}.

Lemma 11. *Let u_1 and u_2 be nodes in REx_1 s.t. u_2 is a descendant of u_1. Then T_{u_2} can be superimposed on T_{u_1}.*

Proof. Let w be a leaf of T_{u_2}. There exists a node $u'_2 \in GST_{D'}$ s.t. $str(u'_2) = str(r'(u_2)) \cdot str(w)$. Since $str(r'(u_1))$ is a suffix of $str(r'(u_2))$, there exists a node $u'_1 \in GST_{D'}$ s.t. $str(u'_1) = str(r'(u_1)) \cdot str(w)$. Thus there also exists a node in T_{u_1} which corresponds to u'_1. □

Because of Lemma 11, $size(T_{u_1}) - size(T_{u_2}) \geq 0$ for any u_1 and u_2 in REx_1 s.t. u_2 is a child of u_1.

The following lemma shows a relation between $cand_1(u)$ and T_u. By using this lemma, we can determine whether $cand_1(u) = \phi$ or not. Now, let $RMax$ be the set of nodes in V_{GST_D} which are d-right-maximal extensions of P. In other words, $RMax$ is the set of answers to the *maximal generic words problem* in [8] for a given pattern P. Let G be the tree which is a subgraph of $GST_D(L(P))$ of which the root is $L(P)$ and the leaves are $RMax$.

Lemma 12. *Let u_1 and u_2 be nodes in REx_1 s.t. u_2 is a child of u_1 and u_2 have no siblings in G. Then $size(T_{u_1}) - size(T_{u_2}) > 0$ iff $cand_1(u_1) \neq \phi$.*

Proof. ($\Rightarrow$). Since $size(T_{u_1}) - size(T_{u_2}) > 0$, there exists some node $u' \in cand(u_1)$ s.t. $Weight_D(str(u')^R \cdot str(u_2)[|str(u_1)| + 1..|str(u_2)|]) < d$. By the definition of $cand$, $str(u')^R$ is *d-left-maximal*. Since u_2 is a child of u_1 and u_2 have no siblings in G, $str(u')^R$ is *d-right-maximal*. Thus $u' \in cand_1(u_1)$. ($\Leftarrow$). We assume $u' \in cand_1(u_1)$. Then there exists a node $w \in V_{T_{u_1}}$ s.t. $str(w) = str(u')[|str(u_1)| + 1..|str(u')|]$. Since $u' \in Occ$, $Weight_D(str(w)^R \cdot str(u_2)) < d$. Thus there doesn't exist a node $v' \in V_{T_{u_2}}$ s.t. $str(v) = str(w)$. Therefore $size(T_{u_1}) - size(T_{u_2}) > 0$. □

Figure 3 shows an example of Lemmas 11, 12.

To use the above lemma, we divide G into paths $\pi_1, \ldots, \pi_s$ s.t. any node which is a child of some node in the same path has no sibling for each path. The paths are defined as follows.

- Each node in G belongs to some path.
- The highest node of each path is a child of a *branching node*.
- The lowest node of each path is a *branching node* or a leaf.
- The other nodes are a *non-branching node*.

Figure 4 shows an example of G and paths.

We compute a node in REx_2 by binary search on each path.

In the rest of this section, we show how to check whether the number of occurrences of P in $str(u)$ is one or more than one. For any node $u \in V_{GST_D}$, let $preord(u)$ be the preorder traversal rank in GST_D, and $[beg(u), end(u)]$ be the interval of the preorder traversal rank of $GST_D(u)$. Obviously, $beg(u) = preord(u)$ holds. Let SA_x be the *suffix*

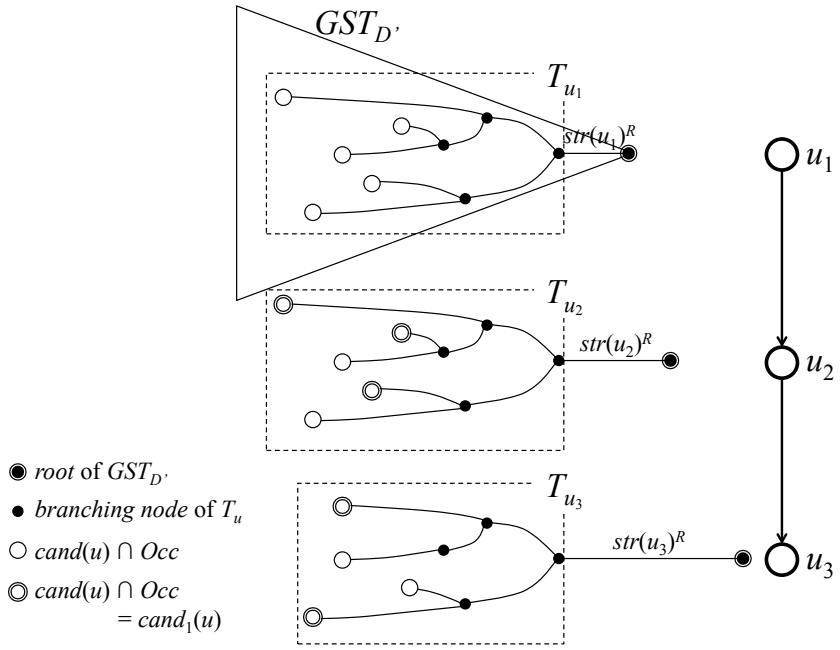

Figure 3. An example of T_{u_1}, T_{u_2}, T_{u_3} s.t. u_1, u_2 and u_3 are successive and non-branching nodes in G. In this example, $size(T_{u_1}) = size(T_{u_2}) > size(T_{u_3})$ since $cand_1(u_1) = \phi$ and $cand_1(u_2) \neq \phi$.

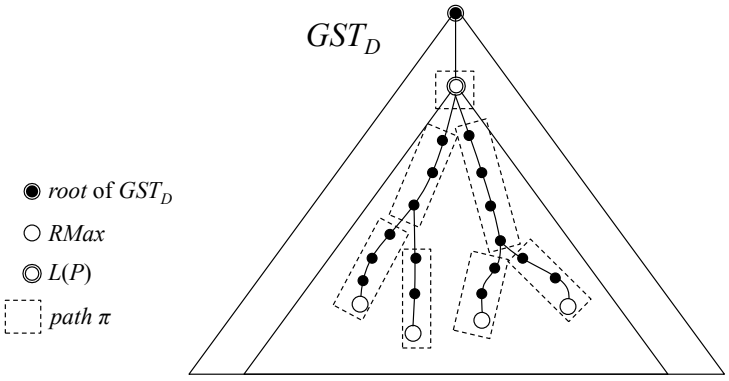

Figure 4. An example of G and its divided paths.

array [9] of a string x, and k be the integer s.t. $SA_{str(u)}[k] = 1$. Let u_s be the node in GST_D s.t. $str(u_s) = str(u)[SA_{str(u)}[k-1]..|str(u)|]$. Let u_ℓ be the node in GST_D s.t. $str(u_\ell) = str(u)[SA_{str(u)}[k+1]..|str(u)|]$. At each node u, we store $[beg(u), end(u)]$ and pointers to u_s and u_ℓ.

Lemma 13. *Let u be a node in $GST_D(L(P))$. $preord(u_s) < beg(L(P)) \leq end(L(P)) < preord(u_\ell)$ iff $str(u)$ has only one occurrence of P (P is a prefix of $str(u)$.).*

Proof. ($\Rightarrow$) Since $preord(u_s) < beg(L(P))$, $str(u_s)$ does not have P as a prefix. Since $end(L(P)) < preord(u_\ell)$, $str(u_\ell)$ does not have P as a prefix. By the definition of u_s and u_ℓ, $str(u)$ have P only as a prefix.

($\Leftarrow$) Since $str(u)$ have P only as a prefix, $str(u_s)$ and $str(u_\ell)$ do not have P as a prefix. By the definition of u_s and u_ℓ, $preord(u_s) < beg(L(P)) \leq end(L(P)) < preord(u_\ell)$ holds. □

4 Algorithm

In this section, we show how to compute *Occ*. We use the lemmas in the previous section. In Section 4.1, we show how to compute $cand_1(u)$ for a given node $u \in REx_2$. In Section 4.2, we show how to compute REx_2. Finally in Section 4.3, we summarize our algorithm.

4.1 Computing $cand_1(u)$

First, we show how to compute $cand_1(u)$ for a given node $u \in REx_1$.

Lemma 14. *There exists a data structure which can compute* $cand_1(u)$ *for any node* $u \in REx_1$ *and any integer* d *in* $O(\log\log n + |cand_1(u)|)$ *time. The size of the data structure is* $O(n)$.

Proof. For each node $u' \in V_{GST_{D'}}$, we represent u' as a horizontal segment $([\max\{maxchild(u'), MFC(u')\} + 1, weight(u')], preord(u'))$. Let $(d, [beg(\ell'_u), end(\ell'_u)])$ be a vertical segment. We use *Segment Intersection Query* for a set of the above horizontal segments [4]. Then a returned horizontal segment corresponds to a node $u' \in GST_{D'}$ s.t. $\max\{maxchild(u'), MFC(u')\} \leq d \leq weight(u')$ and $beg(\ell'_u) \leq preord(u') \leq end(\ell'_u)$. By the definition of $cand_1(u)$, $u' \in cand_1(u)$. Clearly, the number of horizontal segments is $O(n)$. Therefore this lemma holds. □

Using the data structure of the above lemma for a set of horizontal segments, we can compute $cand_1(u)$ for any node $u \in REx_1$. Figure 5 shows an example.

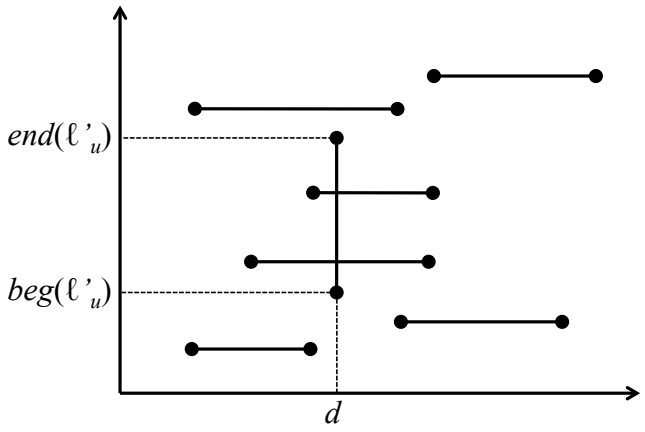

Figure 5. Two horizontal segments which are stabbed by the vertical segment correspond to nodes in $cand_1(u)$.

4.2 Computing REx_2

Second, we show how to compute REx_2.

Lemma 15. *There exists a data structure which can compute* REx_2 *for any node* $L(P) \in V_{GST_D}$ *and any integer* d *in* $O(|REx_2|\log^2 n + |RMax|\log n)$ *time, where* P *is a given string. The size of the data structure is* $O(n\log n)$ *space.*

To prove this lemma, we show some other lemmas. In our algorithm, we use *binary search* based on Lemma 12. The following lemma shows how to compute $size(T_u)$ for any node $u \in REx_1$.

Lemma 16. *For any node $u \in REx_1$, there exists a data structure which can compute $size(T_u)$ in $O(\log n)$ time. The size of the data structure is $O(n \log n)$.*

Proof. By the definition, $size(T_u) = \sum_{u' \in cand(u)} |str(u')| - |cand(u)| \times |str(u)|$. To compute the first term, we use *Segment Intersection Sum Query* as follows. For each node $u' \in V_{GST_{D'}}$, we represent u' as a horizontal segment $([maxchild(u') + 1, weight(u')], preord(u'))$. We define its weight as $|str(u')|$. Let $(d, [beg(\ell'_u), end(\ell'_u)])$ be a vertical segment. Then we can compute the first term in $O(\log n)$ time. Since the sum of the weights is clearly $O(n^2)$, The size of a data structure is $O(n \log n)$ space. On the other hand, we can compute $|cand(u)|$ in $O(\log n)$ time by using *Segment Intersection Count Query* for the above set of horizontal segments and a vertical segment. So we can compute $size(T_u)$ in $O(\log n)$ time with $O(n \log n)$-space data structure. □

To use Lemma 12, we need to compute paths of G. Hence we show that we can compute all branching nodes of G from $RMax$ by the following two lemmas.

Second, we compute branching nodes in G. They are also the lowest node of each path.

Lemma 17. *Given $RMax$, we can compute all branching nodes of G in $O(|RMax| \log |RMax|)$ time.*

Proof. We sort $RMax$ by preorder rank in $O(|RMax| \log |RMax|)$ time. Note that any branching node of G is the lowest common ancestor of two leaf nodes or branching nodes of G. Hence we can compute all branching nodes of G in $O(|RMax|)$ time from sorted $RMax$ by LCA query. □

The following corollary is true by Lemma 13.

Corollary 18. *Given $L(P)$ and a node $u \in V_{GST_D(L(P))}$, we can check in constant time whether $str(u)$ contains P only as a prefix.*

Lemma 19. *Let $\pi = u_1, \ldots, u_k$ denote a path on G s.t. u_1 is a non branching node and the parent node is a branching node of G, u_k is a leaf node or branching node of G and $u_2, \ldots, u_{k-1}$ are non branching nodes of G. Given u_1 and u_k, we can compute all nodes in REx_2 on π in $O((\alpha \log k + 1) \log n)$ time using the data structure of size $O(n \log n)$, where α is the number of the output nodes.*

Proof. Note that we can access any node on π in constant time by level ancestor query. First, we compute the maximum integer $c \leq k$ such that P occurs in $str(u_c)$ only as a prefix by Corollary 18 in $O(\log k)$ time. Second, we check whether there exists at least one node in REx_2 on $u_1, \ldots, u_c$ by comparing $size(T_{u_1})$ and $size(T_{u_c})$ in $O(\log n)$ time. Hence we use the data structures of size $O(n \log n)$ space of Lemma 16. Note that we can compute $r'(u)$ for $u \in V_{GST_D}$ in constant time by preprocessing D in linear space. From Lemma 12, if $size(T_{u_1}) - size(T_{u_c}) > 0$, we compute all nodes in REx_2 on $u_1, \ldots, u_c$ in $O((\alpha \log c + 1) \log n)$ time by binary search. □

Proof of Lemma 15 is the following.

Proof. By Lemma 17. we can compute all branching nodes and leaf nodes of G in $O(|RMax| \log |RMax|)$ time from $RMax$. Note that we can compute $RMax$ in $O(|RMax|)$ time from $L(P)$ using the data structures of size $O(n)$ [8]. By Lemma 19, we can compute all nodes in REx_2 on each path of G in $O((\alpha \log k + 1) \log n)$ time. Hence we can compute REx_2 in $O(|REx_2| \log^2 n + |RMax| \log n)$ time. □

4.3 Overall Complexity

Theorem 20. *There exists a data structure which can solve Problem 1 in* $O(|P| + |Occ| \log^2 n + |RMax| \log n)$ *time. The size of the data structure is* $O(n \log n)$.

Proof. We compute $L(P)$ in $O(|P|)$ time by traversing GST_D. Then we can compute REx_2 by Lemma 15. Finally we compute $cand_1(u)$ for any $u \in REx_2$ by Lemma 14. So the total time complexity is $O(|P| + |Occ| \log^2 n + |RMax| \log n)$. The space requirement of the data structure is $O(n \log n)$. □

5 Conclusion and Future Work

We proposed an $O(n \log n)$-space data structure which can solve the *left-right maximal generic words problem* in $O(|P| + |Occ| \log^2 n + |RMax| \log n)$ time.

Our future work includes the following.

- Can we solve the left-right maximal generic words problem more efficiently? A difficulty of this problem is that a given pattern can be extended to *both* directions.
- When we are given a single text string (a single document), a pattern P, and a threshold d on the number of occurrences of P in the text, is there a simpler algorithm to find the left-right maximal words for this special case?
- In this paper we only considered the maximal generic word problem. In [8], they also considered the minimal discriminating words problem. So the minimal discriminating words problem for the left-right extensions of a given pattern P is also interesting.

References

1. M. A. Bender and M. Farach-Colton: *The LCA problem revisited*, in LATIN 2000: Theoretical Informatics, 4th Latin American Symposium, Punta del Este, Uruguay, April 10–14, 2000, Proceedings, G. H. Gonnet, D. Panario, and A. Viola, eds., vol. 1776 of Lecture Notes in Computer Science, Springer, 2000, pp. 88–94.
2. M. A. Bender and M. Farach-Colton: *The level ancestor problem simplified.* Theor. Comput. Sci., 321(1) 2004, pp. 5–12.
3. S. Biswas, M. Patil, R. Shah, and S. V. Thankachan: *Succinct indexes for reporting discriminating and generic words*, in String Processing and Information Retrieval – 21st International Symposium, SPIRE 2014, Ouro Preto, Brazil, October 20–22, 2014. Proceedings, E. S. de Moura and M. Crochemore, eds., vol. 8799 of Lecture Notes in Computer Science, Springer, 2014, pp. 89–100.
4. T. M. Chan: *Persistent predecessor search and orthogonal point location on the word RAM.* ACM Transactions on Algorithms, 9(3) 2013, p. 22.
5. B. Chazelle: *Filtering search: A new approach to query-answering.* SIAM J. Comput., 15(3) 1986, pp. 703–724.
6. M. Edahiro, K. Tanaka, T. Hoshino, and T. Asano: *A bucketing algorithm for the orthogonal segment intersection search problem and its practical efficiency.* Algorithmica, 4(1) 1989, pp. 61–76.

7. P. Gawrychowski, G. Kucherov, Y. Nekrich, and T. A. Starikovskaya: *Minimal discriminating words problem revisited*, in String Processing and Information Retrieval – 20th International Symposium, SPIRE 2013, Jerusalem, Israel, October 7–9, 2013, Proceedings, O. Kurland, M. Lewenstein, and E. Porat, eds., vol. 8214 of Lecture Notes in Computer Science, Springer, 2013, pp. 129–140.
8. G. Kucherov, Y. Nekrich, and T. A. Starikovskaya: *Computing discriminating and generic words*, in String Processing and Information Retrieval – 19th International Symposium, SPIRE 2012, Cartagena de Indias, Colombia, October 21–25, 2012. Proceedings, L. Calderón-Benavides, C. N. González-Caro, E. Chávez, and N. Ziviani, eds., vol. 7608 of Lecture Notes in Computer Science, Springer, 2012, pp. 307–317.
9. U. Manber and G. Myers: *Suffix arrays: A new method for on-line string searches.* SIAM Journal on Computing, 22(5) 1993, pp. 935–948.
10. C. Sheng and Y. Tao: *New results on two-dimensional orthogonal range aggregation in external memory*, in Proceedings of the 30th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2011, June 12–16, 2011, Athens, Greece, M. Lenzerini and T. Schwentick, eds., ACM, 2011, pp. 129–139.
11. P. Weiner: *Linear pattern-matching algorithms*, in Proc. of 14th IEEE Ann. Symposium on Switching and Automata Theory, Institute of Electrical Electronics Engineers, New York, 1973, pp. 1–11.