Minimal Discriminating Words Problem Revisited

Paweł Gawrychowski[1], Gregory Kucherov[2,3],
Yakov Nekrich[4], and Tatiana Starikovskaya[5]

[1] Max-Planck-Institut für Informatik, Saarbrücken, Germany
gawry@cs.uni.wroc.pl
[2] Laboratoire d'Informatique Gaspard Monge, Université Paris-Est & CNRS, Marne-la-Vallée, Paris, France
Gregory.Kucherov@univ-mlv.fr
[3] Department of Computer Science, Ben-Gurion University of the Negev, Be'er Sheva, Israel
[4] Department of Electrical Engineering & Computer Science, University of Kansas, Lawrence, USA
yakov.nekrich@googlemail.com
[5] School of Applied Mathematics and Information Science, Higher School of Economics, Moscow, Russia
tat.starikovskaya@gmail.com

Abstract. We revisit two variants of the problem of computing *minimal discriminating* words studied in [5]. Given a pattern P and a threshold d, we want to report (i) all shortest extensions of P which occur in less than d documents, and (ii) all shortest extensions of P which occur only in d *selected* documents. For the first problem, we give an optimal solution with constant time per output word. For the second problem, we propose an algorithm with running time $O(|P| + d \cdot (1 + output))$ improving the solution of [5].

1 Introduction

Given a collection of text documents (character sequences), we are often interested in patterns that characterize a certain subset of these documents, i.e., occur only in the documents of this subset and not in the others. Such patterns (*words*) are called *discriminating* with respect to the corresponding subset. Identifying such patterns can be part of a *machine learning* or *data mining* task over a sample of documents, or can arise in *automated text classification*. In computational biology, patterns that appear in a subset of sequences sharing some biological feature and do not appear in the other sequences of the considered sample can be naturally assumed to be responsible for that feature.

In [5], the authors introduced the problem of *minimal discriminating words* along with the complementary problem of *maximal generic words*. In both of them, it is asked to compute some *extensions* of a given pattern P (which can be an empty word), i.e. strings which have P as a prefix. Consider a collection of strings (documents) $T_1, T_2, \ldots, T_m$ of total length n. Two variants of the minimal discriminating words problem have been considered in [5]. The basic variant is to report,

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-3-319-02432-5_33

O. Kurland, M. Lewenstein, and E. Porat (Eds.): SPIRE 2013, LNCS 8214, pp. 129–140, 2013.

given a pattern P and a threshold $d \leq m$, all extensions of P which occur in *at most d* documents and which are minimal, i.e. any proper prefix of a reported extension must occur in more than d documents. A more practically motivated variant, called *minimal discriminating words for specified documents*, is to compute all minimal extensions starting with P which occur only in documents within a given subset $T_{i_1}, T_{i_2}, \ldots, T_{i_d}$. Minimality condition means that any proper prefix of a reported extension must occur in documents other than $T_{i_1}, \ldots, T_{i_d}$.

To exemplify minimal discriminating words, consider $T_1 = baaababb$, $T_2 = babaabab$ and $T_3 = babbaaab$. For $d = 2$, minimal discriminating extensions of $P = aa$ are aaa (discriminates $\{T_1, T_3\}$) and $aaba$ (discriminates $\{T_1, T_2\}$).

The complementary problem of maximal generic words looks for all *maximal* extensions of P occurring in *at least d* documents. In [5] a linear-space solution to the problem of reporting all maximal generic extensions was given. Its running time was optimal time $O(|P| + output)$, where *output* is the number of reported extensions. The same paper proposed efficient solutions for the two variants of the minimal discriminating words problem, but their time bounds were not optimal: the basic variant of the problem was solved in $O(|P| + \log\log n + output)$ time, and the variant with specified documents was solved in $O(|P| + d \log\log m \cdot (1 + output))$ time. Moreover, in the latter case, the solution of [5] has the following undesirable property: it assumes that each T_i ends with a unique sentinel $\$_i$ that can be a part of a discriminating word even if dropping the sentinel yields a word which is not discriminating. Both solutions use $O(n)$ space.

In this paper, we revisit both variants of the minimal discriminating words problem and improve the bounds of [5]. For the second variant, we also get rid of the unnatural assumption about sentinel symbols occurring in discriminating words. Specifically, we propose $O(n)$ space solutions for the first and for the second problem with $O(|P| + output)$ and $O(|P| + d(1 + output))$ time respectively. Thus, for the first variant, we reach the optimal time bound. For the second variant, our running time does not depend on the size nor the number of documents, but only on the number of selected documents. In particular, when this number is constant, we obtain again an optimal $O(|P| + output)$ time. In both cases our solutions have the desirable property that after first spending $O(|P|)$ or $O(|P| + d)$ time, respectively, to initialize the computation, the worst-case delay between reporting two successive extensions is $O(1)$ or $O(d)$, respectively.

Similar to [5], our solutions are based on the generalized suffix tree of $T_1, T_2, \ldots, T_m$ that can be viewed as a compacted trie for strings $T_1\$_1, T_2\$_2, \ldots, T_m\$_m$. It is well-known that the generalized suffix tree can be computed in $O(n)$ time. For each node v of the generalized suffix tree we store its weight $\mathsf{weight}(v)$ defined as the number of *distinct* documents whose suffixes occur in the subtree rooted at v. All $\mathsf{weight}(v)$ can be computed in $O(n)$ time [3].

The *locus* of a string S in a trie on a set of strings is defined as the highest explicit node labelled by an extension of S. It is important to note that in all our algorithms, each output word is specified by its locus in the generalized suffix tree for $T_1, T_2, \ldots, T_m$ (rather than by "spelling out" the word itself).

2 Minimal Discriminating Words

Suppose that a set of documents $T_1, T_2, \ldots, T_m$ of total length n is given. For a pattern P and a threshold $d \leq m$, we want to find all minimal extensions of P which occur in at most d distinct documents. "Minimal" here means that no proper prefix of a reported extension satisfies this property. We describe a linear-space data structure for this problem.

2.1 General Idea

Consider the generalized suffix tree for $T_1, T_2, \ldots, T_m$. We first delete sentinels $\$_i$ from labels of its edges, and then delete edges with empty labels. Consider the locus of P in the resulting trie, which we call GST. Any descendant u of the locus such that $\mathsf{weight}(u) \leq d$ and $\mathsf{weight}(p(u)) > d$, where $p(u)$ is the parent of u, will be a locus of a desired extension of P. The extension itself will be equal to the label of $p(u)$ extended by the first letter on the edge $(p(u), u)$. (By construction the first letter on any edge of GST is not a sentinel but a letter of the alphabet).

We represent GST with its compacted version GST^c, and a number of arrays. An array A_u corresponding to an edge $(p(u), u)$ of GST^c contains links to nodes which were removed in order to obtain the edge $(p(u), u)$. More precisely, $A_u[\Delta]$ links to the lowest ancestor v of u such that $\mathsf{weight}(v) \geq \mathsf{weight}(u) + \Delta$, for every $\Delta < \mathsf{weight}(p(u)) - \mathsf{weight}(u)$. Note that the total length of the arrays is just $O(n)$ as each entry corresponds to one suffix. Loci of the extensions can be found in the following way: first, find the locus v of P in GST^c and compute all nodes u in its subtree for which $\mathsf{weight}(u) \leq d$ and $\mathsf{weight}(p(u)) > d$. Then for each found node u compute its ancestor u' in GST such that $\mathsf{weight}(u') \leq d$ and $\mathsf{weight}(p(u')) > d$. The last step can be done in constant time using the array A_u associated with $(p(u), u)$: we choose as u' the node $A_u[d - \mathsf{weight}(u)]$, and if $\mathsf{weight}(u') > d$ we replace it by the unique son on the $u - p(u)$ path. Then the node u' will be a locus of a desired extension. We refer to nodes u as above as *extension loci*.

Let v be the locus of P in GST^c. We denote the subtree of GST^c rooted at v by T_v. Leaves of GST^c of weight bigger than d will be called *d-heavy leaves*. First note that if T_v has no d-heavy leaves, every root-to-leaf path contains an extension locus and hence we can find each extension locus in *amortized* constant time by traversing T_v in depth-first order. Below we explain how to overcome the assumption about d-heavy leaves and to achieve *worst-case* constant time per an extension locus. We first prove the following useful lemma.

Lemma 1. *Each extension locus belongs to a maximal subtree of* T_v *without d-heavy leaves.*

Proof. An extension locus u cannot have a d-heavy leaf in its subtree, otherwise weight of u would be bigger than d. Let u' be the highest ancestor on the path from u to v that does not have a d-heavy leaf in its subtree. Then u belongs to $\mathsf{T}_{u'}$, and $\mathsf{T}_{u'}$ is a maximal subtree of T_v that does not have d-heavy leaves. □

The algorithm will iterate over the maximal subtrees of T_v without d-heavy leaves and report extension loci for each of them. We give the details below.

2.2 Computing Maximal Subtrees

Let a trie τ_k, $0 \leq k \leq m-1$, be a compact trie containing labels of all k-heavy leaves. (Note that τ_0 is essentially GST^c.) From the construction it follows that there is one-to-one correspondence between leaves of τ_k and k-heavy leaves. Moreover, for each node u of τ_k there is a node w of GST^c such that the labels of u and w are equal. Such nodes w will be referred to as *k-nodes*. We say that u and w are of type 1 iff the degree of u is smaller than the degree of w, and that they are of type 2 iff there is at least one node on the path from w to its nearest k-node ancestor. (A node can be of type 1 and of type 2 simultaneously or neither of type 1 nor of type 2.) We store nodes of types 1 and 2 in two lists ordered as in the depth-first traversal of τ_k. Next, let us consider a node of GST. Nodes of type 1 in its subtree form a sublist in the first list. For each node we store pointers to the start and to the end of the corresponding sublist. Pointers associated with nodes of type 2 are defined in a similar way.

Note that the parent p of the root of a maximal subtree T without d-heavy leaves has a d-heavy leaf in its subtree (otherwise, T would not be maximal). That is, p is either a d-node or a node on the path connecting a d-node and its nearest d-node ancestor. At the same time, p has at least one son (the root of T) which does not have a d-leaf, and, consequently, a d-node, in its subtree. Therefore, in the first case p is a d-node of type 1, and in the second case p is on the path connecting a d-node of type 2 and its nearest d-node ancestor.

We now return to the description of the algorithm. We start by computing the locus of P in τ_d in $O(|P|)$ time in a usual way. Then we iterate over nodes of types 1 and 2 in the subtree of the locus using the pointers and the lists and report associated maximal subtrees without d-heavy leaves.

Let w be a d-node of GST^c of type 1, and u be the corresponding node of τ_d. By the definition, the degree of u is smaller than the degree of w, which means that at least one child of w is a root of a maximal subtree without d-heavy leaves. Such children form subranges of the list of all children of w, and we assume that pointers to these subranges are available. (As we show below, the total number of the pointers is linear and they can be precomputed in linear time.) Using the pointers we can output i requested children of w in $O(i)$ time.

If w is a d-node of GST^c of type 2 and w' is its nearest d-node ancestor, then all subtrees hanging off the path from w to w' are maximal subtrees without d-heavy leaves. The subtrees can be found in linear time by iterating over nodes on the path from w to w'.

All in all, retrieving maximal subtrees without d-heavy leaves takes constant time per subtree in the worst case.

Lemma 2. *Tries τ_k and pointers to the subranges of children of k-nodes that do not have k-heavy leaves in their subtrees occupy $O(n)$ space in total and can be constructed in $O(n)$ time.*

Proof. To estimate the space occupied by the tries it is enough to estimate the total number of their leaves. The latter is equal to n, because a string which is a suffix of k documents will correspond to a leaf in τ_1, to a leaf in τ_2, ..., to a leaf in τ_k, that is, k leaves in total. The statement follows.

The tries are built as follows. We first augment GST with a linear-space data structure [8] that allows to answer lowest common ancestor queries in constant time. This step takes $O(n)$ time. We then iterate over leaves of GST^c from the left to the right and for each k compose a lexicographically ordered list L_k of k-heavy leaves' labels. Secondly, we scan L_k and compute the length of the longest common prefix of every two consecutive suffixes in L_k. (The length is equal to the string depth of the lowest common ancestor of the leaves corresponding to the suffixes and hence can be computed in constant time). Once we have L_k and the lengths, we build τ_k in linear time in a usual way. Correspondence between nodes of τ_k and GST^c and hence types of nodes can be established in $O(|\tau_k|)$ time with the help of the lowest common ancestor queries. Finally, the lists of nodes of types 1 and 2 are constructed by depth-first traversal of τ_k.

Let w be a k-node of GST^c and u be the corresponding node of τ_k. The number of subranges formed by children of w without k-heavy leaves in their subtrees does not exceed the degree of u. Therefore, the total number of the subranges does not exceed the total size of the tries, which is $O(n)$. Next, note that a node does not have k-heavy leaves in its subtree if and only if the weight of the heaviest leaf in its subtree $\leq k$. We compute the subranges in two steps. First we traverse GST^c bottom-up and for each node compute the weight of the heaviest leaf in its subtree. Secondly, we scan the list of children of each node and for each k such that the node is a k-node remember the starting and the ending points of maximal subranges with the weights $\leq k$. Construction takes linear time in total. □

2.3 Computing Extension Loci

Here we show how to report all extension loci in a maximal subtree of T_v without d-heavy leaves. We start with an auxiliary lemma.

Lemma 3. *A compact trie of size n can be partitioned into disjoint node-to-leaf paths of length $O(\log n)$ each.*

Proof. For a node u of the trie we define $h(u)$ to be the length of the shortest downward path to a leaf from u, and $\ell(u)$ to be the number of leaves in the subtree rooted at u. We prove by induction that $\ell(u) \geq 2^{h(u)}$.

If $h(u) = 0$, then u is a leaf and $\ell(u) = 1$. Suppose that the inequality holds for all u such that $h(u) \leq k$. A node u of the compact trie with $h(u) = k + 1$ has at least two descendants v_1, v_2 and both $h(v_1)$ and $h(v_2)$ must be at least k, hence $\ell(u) \geq 2^{h(v_1)} + 2^{h(v_2)} \geq 2^{k+1}$, the claim follows.

For each node u of the compact trie we colour the edge from u to its child v with the smallest $h(u)$ red. This colouring induces a partition of all nodes into node-disjoint red paths. From the inequality it follows that the length of any red path is $O(\log n)$. □

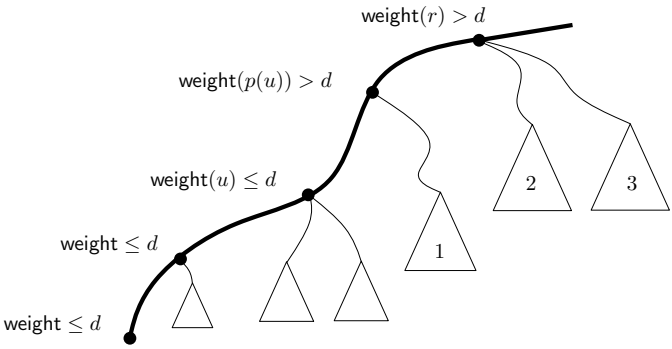

Fig. 1. We push the root of the subtree 1 into S first. When we pop it from S, we push the root of the subtree 2 into S, and so on.

We partition GST^c into disjoint node-to-leaf paths of length $O(\log n)$ using Lemma 3. q-heaps [4] allow to support predecessor queries on logarithmic-size subsets of $[1, n]$ in constant time using linear space and a common precomputed table of size $o(n)$. We use q-heaps to answer predecessor queries on weights of each path of the partition. The total space occupied by q-heaps is $O(n)$.

Lemma 4. *Given a maximal subtree* T *of* T_v *without* d*-heavy leaves, all extension loci in* τ *can be reported in* $O(1)$ *time per locus.*

Proof. To simplify the description, assume that nodes of GST^c are rearranged so that an edge from a node to its leftmost child is always red.

Let r be the root of T. If $\mathsf{weight}(r) \leq d$, then the only extension locus in the subtree is r. (Remember that the parent of r has a d-heavy leaf in its subtree and therefore its weight is bigger than d). If $\mathsf{weight}(r) > d$, we start with the node-to-leaf path containing r. Since the weight of any leaf of T is at most d, the path contains an extension locus u. Using one predecessor query, we can find u in $O(1)$ time. We then push the second child of a parent of u into a stack S.

We perform recursive calls for the subtrees rooted at nodes from S. Each time we pop a node w from S we push its right brother into S. If no brothers are left and the parent and the grandparent of w are on the same red path, we push the second child of the grandparent of w into S (see Fig. 1). The algorithm stops when S is empty.

We now show that the algorithm is correct. Note that any subtree hanging off the path below u does not contain extension loci, while each tree hanging off the path above u contains at least one such node. All the latter trees are examined due to the order of recursive calls. Each call takes constant time and returns a requested node. □

To sum up, each extension locus inside a given maximal subtree without d-heavy leaves can be reported in constant time in the worst case. Since, according to Section 2.2, each such subtree of T_v can be identified in worst-case constant time, we obtain the final theorem.

Theorem 1. *For a given pattern P and a threshold d, all minimal discriminating extensions of P can be reported in time $O(|P| + output)$, where output is the number of reported extensions. The underlying indexing data structure occupies $O(n)$ space, where n is the total length of the strings $T_1, T_2, \ldots, T_m$.*

3 Minimal Discriminating Words for Specified Documents

In many applications, we need to compute words that discriminate documents from a *given sample*. Consider a set of documents $T_1, T_2, \ldots, T_m$ of total length n. Given a set of indices $\mathsf{Ind} = \{i_1, i_2, \ldots, i_d\}$ and a pattern P, we want to find all minimal extensions of P occurring *only* in documents $T_i, i \in \mathsf{Ind}$, where "minimal" means that any of their proper prefixes has at least one occurrence in a document which does not belong to this subset.

Here we propose a linear-space data structure which allows to compute such extensions in time $O(|P| + d \cdot (output + 1))$, where *output* is the number of reported extensions.

3.1 General Idea

Consider the generalized suffix tree for $T_1, T_2, \ldots, T_m$. For each suffix of $T_1, T_2, \ldots, T_m$ we create an explicit node labelled by this suffix (if it does not exist already). We denote the resulting tree by GST. Note that the size of GST is $O(n)$. Problems of computing loci of the minimal extensions in the generalized suffix tree and GST are equivalent.

An inner node of GST is called $\$$-*terminating* if all its outgoing edges are labelled by sentinels. If, in addition, the sentinels are $\$_i$, where $i \in \mathsf{Ind}$, then the node is called Ind-*terminating*. From the definition it follows that the locus of any string occurring only in documents T_i, $i \in \mathsf{Ind}$, contains an Ind-terminating node in its subtree, in particular, the locus of any minimal extension contains such node in its subtree. Besides, each Ind-terminating node belongs to a subtree rooted at the locus of some minimal extension, as shown below.

Lemma 5. *Suppose that w is an Ind-terminating node and that its label starts with P. Then the path from w to the root contains a locus of a minimal extension of P occurring only in documents $T_i, i \in \mathsf{Ind}$.*

Proof. The label S of w is an extension of P occurring only in documents T_i, $i \in \mathsf{Ind}$. The locus of the shortest prefix of S occurring only in documents T_i, $i \in \mathsf{Ind}$, will be the locus of a requested extension and will be on the path from w to the root of GST. □

A high-level description of the algorithm is as follows. We start by locating the locus u of P in GST in time $O(|P|)$ and retrieving the interval $[L(u), R(u)]$ of ranks of suffixes ending below u. Rank of a suffix is simply its rank in the lexicographic order, equal suffixes are assigned equal ranks. The algorithm keeps

a stack of intervals which it is to process, initialized to contain just $[L(u), R(u)]$. At each step it pops an interval $[a, b]$ from the stack, finds an lnd-terminating node v covering a subrange of $[a, b]$, computes the ancestor w of v labelled by a requested extension of P, and pushes the intervals $[a, L(w) - 1]$ and $[R(w) + 1, b]$ onto the stack. If there is no such lnd-terminating node, the algorithm does nothing. The algorithm terminates when the stack is empty.

To estimate the running time of the algorithm, we note that each of the processed intervals, except for $[L(u), R(u)]$, either corresponds to a reported extension, or is a child of an interval corresponding to a reported extension (and each such interval has two children). Hence the total number of processed intervals will be $O(output + 1)$, where *output* is the number of reported extensions. Below we show that processing of each interval takes $O(d)$ time. Note that if we want to make sure that the delay between reporting two minimal extensions is $O(d)$, we only need to check if the interval contains an lnd-terminating node before we push it onto the stack.

3.2 Computing an lnd-Terminating Node

Given an interval $[a, b]$, we want to find some lnd-terminating node u such that all leaves in its subtree are of ranks in $[a, b]$, or to show that there is none. Below we show that it can be done in $O(d)$ time.

Consider a trie T on the reverses $T_1^R, T_2^R, \ldots, T_m^R$ of the documents. Each node v of T corresponds to a prefix of some T_j^R, or, equivalently, to a reversed suffix of T_j. We call the node v *active* if the suffix is a label of a \$-terminating node of GST. If the node is also lnd-terminating, we call v lnd-good, otherwise we call it lnd-bad. Note that if a node is lnd-bad, then all its ancestors are lnd-bad. That is, lnd-good nodes are exactly active nodes of maximal subtrees of T without lnd-bad nodes. We compactify T leaving nodes labelled by T_i^R, $1 \leq i \leq m$, explicit. The resulting trie is denoted by T^c (see Fig. 2).

For an edge e of T^c we define a set $S(e)$ to contain ranks of some suffixes of $T_1, T_2, \ldots, T_m$ in the lexicographic order. The suffixes are exactly the suffixes

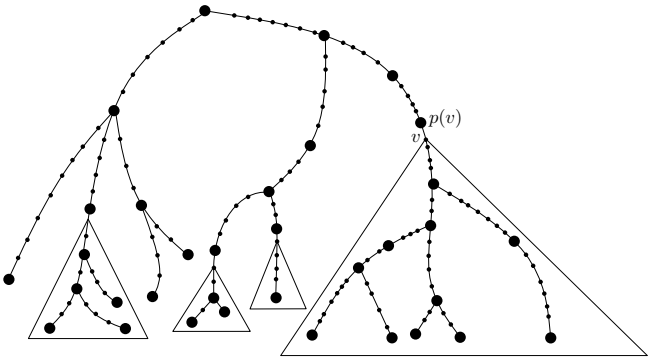

Fig. 2. Maximal subtrees of T without any lnd-bad nodes. Thick nodes exist in T^c.

the reverses of which were the labels of the active nodes removed in order to obtain e.

Lemma 6 (Theorem 4 in [1]). *$S(e)$ can be stored using linear space so that given any interval $[a, b]$ we can in $O(1)$ time either retrieve some element in $S(e) \cap [a, b]$ or detect that there is none.*

Remember that we want to find an Ind-terminating node of GST such that all leaves in its subtree are of ranks in $[a, b]$. The algorithm will search, instead, for an active node of maximal subtrees of T without Ind-bad nodes corresponding to such Ind-terminating node.

Consider a maximal subtree of T without Ind-bad nodes (see Fig. 2). The parent $p(v)$ of its root is labelled by a prefix of T_j^R, for some $j \notin \mathsf{Ind}$, while v is not. Hence, $p(v)$ is either a node of degree bigger than 1, or is labelled by T_j^R. In both cases, $p(v)$ is a node of T^c. It follows that we can decompose a set of active nodes of the subtree into a set of active nodes existing in T^c and sets of active nodes associated with the edges of T^c. Each leaf v of a maximal subtree without Ind-bad nodes corresponds to T_i^R, for some $i \in \mathsf{Ind}$. We will make use of precomputed values $\mathsf{lcp}(i)$, where $\mathsf{lcp}(i)$ is the length of the longest common prefix of T_i^R and T_j^R, $j \notin \mathsf{Ind}$. Note that all ancestors of v of string depth bigger than $\mathsf{lcp}(i)$ belong to the maximal subtree.

We start at a node v labelled by some T_i^R, $i \in \mathsf{Ind}$, and go up until we reach a node of string depth $\mathsf{lcp}(i)$ or an already visited node. For each encountered node we check if it is active and if it corresponds to the desired Ind-terminating node. If yes, the algorithm stops. For each edge e we traverse we try to retrieve an element in $S(e) \cap [a, b]$. If there is such an element, the algorithm finds the corresponding Ind-terminating node and stops. We repeat such procedure for each $i \in \mathsf{Ind}$.

The total number of processed nodes and edges is bounded by the total size of the maximal subtrees without Ind-bad nodes. Since each leaf and each inner node of degree one in the subtrees corresponds to T_i^R, $i \in \mathsf{Ind}$, the total size of the subtrees, and hence the time of computing an Ind-terminating node covering a subrange of $[a, b]$, is $O(d)$. It remains to show that the values $\mathsf{lcp}(i)$, $i \in \mathsf{Ind}$, can be precomputed efficiently.

Lemma 7. *Given Ind, we can compute $\mathsf{lcp}(i)$ for all $i \in \mathsf{Ind}$ in $O(d)$ total time.*

Proof. To compute the values, we use an array R defining the lexicographic order on $T_1^R, T_2^R, \ldots, T_m^R$, its inverse R^{-1} and an array LCP which contains the length of the longest common prefix of every pair of consecutive (in the lexicographic order) reversed documents. The LCP array is augmented with a range minimum query data structure [2], which allows to compute the minimum value in any interval of LCP in constant time. All these structures are built in the preprocessing phase without knowing Ind.

Consider an index $i \in \mathsf{Ind}$, and let $T_{k_1}^R$ and $T_{k_2}^R$ with $k_1, k_2 \notin \mathsf{Ind}$ be the reversed documents closest to T_i^R in the lexicographic order from the left and from the right, respectively. From the properties of the lexicographic order it

follows that $\mathsf{lcp}(i)$ is equal to the maximum of the lengths of the longest common prefixes of $T^R_{k_1}$ and T^R_i and of $T^R_{k_2}$ and T^R_i. The lengths can be computed by taking the minimum of the values stored in the array LCP between the entries corresponding to $T^R_{k_1}$ and T^R_i and of $T^R_{k_2}$ and T^R_i respectively. Hence the only question is how to find k_1 and k_2 efficiently.

Consider the occurrences of T^R_i for all $i \in \mathsf{Ind}$ in R, and let $R[a..b]$ be a maximal interval of such occurrences, i.e., both $R[a-1]$ and $R[b+1]$ correspond to reversals outside of Ind. Then $k_1 = a - 1$ and $k_2 = b + 1$ for all $i \in \mathsf{Ind}$ corresponding to occurrences in the interval. Our method identifies such maximal intervals one-by-one and updates the values of $\mathsf{lcp}(i)$ accordingly. To make the identification efficient, we store an additional bit vector B of length m to keep track of the already processed indices from Ind, initially containing all zeros. We loop over Ind, and if a given $i \in \mathsf{Ind}$ is not processed yet, sweep to the left and to the right starting from $R^{-1}[i]$ to identify the maximal interval of R containing i and some other indices from Ind. Then knowing the values of k_1 and k_2 for all indices in the interval we calculate their values of lcp. Finally, we set their corresponding bits in B to one. In the very end we iterate through all $i \in \mathsf{Ind}$ and clear their corresponding bits in B. The algorithm clearly spends just constant time per a single element of Ind. □

3.3 Computing Ancestor Loci

Here we show how to find the ancestor w of an Ind-terminating node that corresponds to a minimal discriminating word. We assume that for each node v of GST there is a pointer to its highest ancestor with the same weight and that the ranks $L(v)$ and $R(v)$ of the leftmost and the rightmost leaves in the subtree of v can be retrieved in $O(1)$ time. We also store an array D such that $D[i] = k$ if the i-th leaf of GST in the left-to-right order corresponds to a suffix from T_k.

Lemma 8. *Given a node u in GST and $M \in [L(u), R(u)]$, for all distinct values j occurring in $D[L(u), R(u)]$ we can find the leftmost occurrence of j after position M and the rightmost occurrence of j before M in $D[L(u), R(u)]$ in $O(\mathsf{weight}(u))$ time, where $\mathsf{weight}(u)$ is the number of* distinct *documents whose suffixes occur in the subtree rooted at u.*

Proof. We can enumerate all distinct values in an interval of D using the data structure of Muthukrishnan [7]. As follows from the description in [7], the structure reports the leftmost occurrence of each j that occurs in the interval. By reversing the input, we can modify the structure so that the rightmost occurrence of each j is reported, too. We obtain the result by reporting the leftmost occurrence of each distinct j in the interval $D[M, R(u)]$ and the rightmost occurrence of each j in the interval $D[L(u), M]$. □

In Lemma 5 we showed that any Ind-terminating node w in the subtree rooted at the locus of P has an ancestor v that is a locus of a desired minimal extension. We compute v in two steps.

Using the pointers we can find the highest ancestor w' of w of weight at most d in $O(d)$ time. The interval $D[L(w'), R(w')]$ contains indices of at most d different documents, and we output these indices in time $O(d)$ using Lemma 8. For each index j that occurs in $D[L(w'), R(w')]$ but does not belong to Ind, we find the rightmost occurrence of j before $L(w)$. The maximum (rightmost) position among them is denoted by L'. Similarly, we find the leftmost position of each $j \notin \mathsf{Ind}$ after $R(w)$, and denote the leftmost among them by R'. This step takes $O(d)$ time. $[L', R']$ is the maximal segment that contains $[L(w), R(w)]$ and consists only of indices from Ind.

Now consider the node w again. We initialize v to w and jump from v to the highest node v' such that $\mathsf{weight}(v) = \mathsf{weight}(v')$. Let $p(v')$ be the parent of v'. If $L' \le L(p(v')) \le R(p(v')) \le R'$, we set $v = p(v')$ and repeat the same step for the new node v. Otherwise we set $v = v'$ and stop. Observe that each iteration increases the number of different indices occurring in $[L(v), R(v)]$ by at least one and takes just constant time.

Lemma 9. *Given an* Ind*-terminating node* w *in the subtree of* u *being the locus of* P*. The node on the path from* w *to the root of* GST *that is a locus of a minimal extension of* P *occurring only in documents* T_i*,* $i \in \mathsf{Ind}$*, can be computed in* $O(d)$ *time.*

Combining Lemma 9 and the algorithm described in Section 3.2, we obtain the final result.

Theorem 2. *Given a subset of indices* $\{i_1, i_2, \ldots, i_d\}$ *and a pattern* P*, all minimal extensions of* P *which occur only in the documents* $T_{i_1}, T_{i_2}, \ldots, T_{i_d}$ *can be computed in time* $O(|P| + d(output + 1))$*, where output is the number of reported extensions. The underlying indexing data structure occupies* $O(n)$ *space, where* n *is the total length of the strings* $T_1, T_2, \ldots, T_m$*.*

We can also output the loci of minimal extensions in lexicographic order without increasing the query time. We achieve this by keeping intervals $[L(w), R(w)]$ for all found extension loci w in a tree $\mathcal{T}$. We initialize $\mathcal{T}$ to a one-node tree and store the interval $[L(u), R(u)]$ at its root r. If we find a new extension locus w, we replace $[L(u), R(u)]$ with $[L(w), R(w)]$ and append two child nodes to r. Intervals $[L(u), L(w) - 1]$ and $[R(w) + 1, R(u)]$ are stored in the left and the right children of r respectively. Every time when we find an Ind-terminating node v in the interval $[l(\nu), r(\nu)]$ stored in some $\nu \in \mathcal{T}$, we identify the ancestor w of v that is the locus of a minimal extension. Then we replace $[l(\nu), r(\nu)]$ with $[L(w), R(w)]$ and append two child nodes to ν as described above. When all loci are found, we traverse internal nodes of $\mathcal{T}$ in-order to obtain a sorted list L of the intervals $[L(w), R(w)]$ for extension loci w. The traversal of $\mathcal{T}$ takes $O(output)$ time; thus the total asymptotic time necessary to answer a query remains unchanged.

We remark that the data structure of Theorem 2 can be constructed in $O(n \log^{\varepsilon} n)$ time for any constant $\varepsilon > 0$ with high probability [6]. The preprocessing time is dominated by the cost of constructing data structures $S(e)$.

4 Conclusions

We developed an optimal algorithm for reporting all minimal discriminating words. For the problem of reporting all minimal discriminating words for a specified set of documents, our solution is optimal when $d = O(1)$, but it might still be possible to improve the running time for the case of non-constant value of d.

Another interesting question is whether counting the number of solutions can done faster than reporting them all according to our algorithm. Finally, we also wonder if we can generate k lexicographically smallest solutions in time proportional to k rather than to *output*. Our algorithms can be used to output k distinct solutions with such complexity, but we cannot guarantee that the generated solutions are lexicographically smallest.

References

1. Alstrup, S., Brodal, G.S., Rauhe, T.: Optimal static range reporting in one dimension. In: Proc. of the 33rd Annual ACM Symposium on Theory of Computing, pp. 476–482 (2001)
2. Bender, M.A., Farach-Colton, M.: The LCA problem revisited. In: Gonnet, G.H., Viola, A. (eds.) LATIN 2000. LNCS, vol. 1776, pp. 88–94. Springer, Heidelberg (2000)
3. Hui, L.C.K.: Color set size problem with applications to string matching. In: Apostolico, A., Galil, Z., Manber, U., Crochemore, M. (eds.) CPM 1992. LNCS, vol. 644, pp. 230–243. Springer, Heidelberg (1992)
4. Fredman, M.L., Willard, D.E.: Trans-dichotomous algorithms for minimum spanning trees and shortest paths. J. Comput. Syst. Sci. 48(3), 533–551 (1994)
5. Kucherov, G., Nekrich, Y., Starikovskaya, T.: Computing discriminating and generic words. In: Calderón-Benavides, L., González-Caro, C., Chávez, E., Ziviani, N. (eds.) SPIRE 2012. LNCS, vol. 7608, pp. 307–317. Springer, Heidelberg (2012)
6. Mortensen, C.W., Pagh, R., Patrascu, M.: On dynamic range reporting in one dimension. In: Proc. of the 37th Annual ACM Symposium on Theory of Computing, pp. 104–111 (2005)
7. Muthukrishnan, S.: Efficient algorithms for document retrieval problems. In: Proc. of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms. SIAM (2002)
8. Schieber, B., Vishkin, U.: On finding lowest common ancestors: Simplification and parallelization. SIAM Journal on Computing 17, 111–123 (1988)