Cross-Document Pattern Matching

Tsvi Kopelowitz, Gregory Kucherov, Yakov Nekrich, Tatiana Starikovskaya

Cross-Document Pattern Matching[☆]

Tsvi Kopelowitz

Weizmann Institute of Science, Rehovot, Israel

Gregory Kucherov

Laboratoire d'Informatique Gaspard Monge, Université Paris-Est &CNRS, Marne-la-Vallée, Paris, France

Yakov Nekrich[1]

Department of Electrical Engineering & Computer Science, University of Kansas, Lawrence, USA

Tatiana Starikovskaya[2]

Lomonosov Moscow State University, Moscow, Russia

Abstract

We study a new variant of the pattern matching problem called *cross-document pattern matching*, which is the problem of indexing a collection of documents to support an efficient search for a pattern in a selected document, where the pattern itself is a substring of another document. Several variants of this problem are considered, and efficient linear space solutions are proposed with query time bounds that either do not depend at all on the pattern size or depend on it in a very limited way (doubly logarithmic). As a side result, we propose an improved solution to the *weighted ancestor* problem.

Keywords: algorithms; pattern matching, document reporting; weighted

[☆]A preliminary version of this work has been presented at the 23rd Annual Symposium on Combinatorial Pattern Matching in July 2012.

[1]This work was done while this author was at Laboratoire d'Informatique Gaspard Monge, Université Paris-Est &CNRS, Marne-la-Vallée, Paris, France

[2]Corresponding author.

[3]E-mail addresses: kopelot@gmail.com (T. Kopelowitz), Gregory.Kucherov@univ-mlv.fr (G. Kucherov), yakov.nekrich@googlemail.com (Y. Nekrich), tat.starikovskaya@gmail.com (T. Starikovskaya)

ancestor problem.

1. Introduction

In this paper we study the following variant of the pattern matching problem that we call *cross-document pattern matching*: given a collection of strings (documents) stored in a "database", we want to be able to efficiently search for a pattern in a given document, where the pattern itself is a substring of another document. More formally, assuming we have a set of documents $T_1, T_2, \ldots, T_m$, we want to answer queries about the occurrences of a substring $T_k[i..j]$ in a document T_ℓ.

This scenario may occur in various situations when we have to search for a pattern in a text stored in a database, and the pattern is itself drawn from a string from the same database. In bioinformatics, for example, a typical project deals with a selection of genomic sequences, such as a family of genomes of evolutionary related species. A common repetitive task consists then in looking for genomic elements belonging to one of the sequences in some other sequences. These elements may correspond to genes, exons, mobile elements of any kind, regulatory patterns, etc., and their location (i.e. start and end positions) in the sequence of origin is usually known from a genome annotation provided by a sequence data repository (such as GenBank or any other). A similar scenario may occur in other application fields, such as the bibliographic search for example.

In this paper, we study different versions of the cross-document pattern matching problem. First, we distinguish between counting and reporting queries, asking respectively about the number of occurrences of $T_k[i..j]$ in T_ℓ or about the occurrences themselves. The two query types lead to slightly different solutions. In particular, the counting problem uses the *weighted ancestor* problem [9, 20, 2] to which we propose a new solution with an improved complexity bound.

We further consider different variants of the two problems. The first one is the dynamic variant where new documents can be added to or deleted from the database. In another variant, called *document counting and reporting*, we only need to respectively count or report the documents containing the pattern, rather than counting or reporting pattern occurrences within a given document. This version is very close to the *document retrieval problem* previously studied (see [22] and later papers referring to it), with the difference that in our case the pattern is itself selected from the documents stored in the database. Finally, we also consider *succinct* data structures

for the above problems, where we keep the underlying index data structure in compressed form. In particular, we propose a generic solution to the weighted ancestor problem on a compressed suffix tree, whose running time is expressed in terms of time bounds for suffix tree operations. This generic solution is then applied to obtain a compact-space solution of the document reporting problem.

Let m be the number of stored strings and n the total length of all strings. Our results are summarized below.

(I) For the counting problem, we propose a solution with query time $O(t + \log\log m)$, where $t = \min(\sqrt{\log occ / \log\log occ}, \log\log |P|)$, $P = T_k[i..j]$ is the searched substring and occ is the number of its occurrences in T_ℓ.

(II) For the reporting problem, our solution outputs all the occurrences in time $O(\log\log m + occ)$.

(III) In the dynamic case, when documents can be dynamically added or deleted, we are able to answer counting queries in time $O(\log\log n)$ and reporting queries in time $O(\log\log n + occ)$, whereas the updates take $O(\log\log n + \log\log \sigma)$ expected time per character, where σ is the size of the alphabet.

(IV) For the document counting and document reporting problems, our algorithms run in time $O(\log n)$ and $O(t + ndocs)$ respectively, where t is as above and $ndocs$ is the number of reported documents.

(V) Finally, we also present succinct data structures that support counting, reporting, and document reporting queries in the cross-document scenario (see Theorems 6 and 7 in Section 4.3).

For problems (I)-(IV), the involved data structures occupy $O(n)$ space under the RAM model. Interestingly, in the cross-document scenario, the query times either do not depend at all on the pattern length or depend on it in a very limited (doubly logarithmic) way.

The paper is organized as follows. In Section 2.1, we briefly introduce main data structures used in the paper and then, in Section 2.2, describe an improved solution to the weighted ancestor problem that we use in the following sections. Section 3 is devoted to counting and reporting versions of the basic cross-document pattern matching problem. In Section 4, we study different variants of the basic problem. First, in Section 4.1, we focus on the dynamic version, when documents can be dynamically added or

deleted. Then, in Section 4.2, we turn to the document counting and reporting versions, when we only seek documents containing the pattern and not occurrences themselves. Finally, in Section 4.3 we propose solutions to the basic problems using succinct data structures.

Throughout the paper positions in strings are numbered from 1. Notation $T[i..j]$ stands for the substrings $T[i]T[i+1]\dots T[j]$ of T, and $T[i..]$ denotes the suffix of T starting at position i.

2. Preliminaries

2.1. Basic Data Structures

We assume a basic knowledge of suffix trees and suffix arrays.

Besides using suffix trees for individual strings T_i, we will also be using the *generalized suffix tree* for a set of strings $T_1, T_2, \dots, T_m$ that can be viewed as the suffix tree for the string $T_1\$_1T_2\$_2\dots T_m\$_m$. A leaf in the suffix tree for T_i is associated with a distinct suffix of T_i, and a leaf in the generalized suffix tree is associated with a suffix of some document T_i together with the index i of this document. We assume that for each node v of a suffix tree, the number n_v of leaves in the subtree rooted at v, as well as its string depth $d(v)$ can be recovered in constant time. Recall that the string depth $d(v)$ is the total length of strings labeling the edges along the path from the root to v.

We will also use suffix arrays for individual documents as well as the *generalized suffix array* for strings $T_1, T_2, \dots, T_m$. Each entry of the suffix array for T_i is associated with a distinct suffix of T_i and each entry of the generalized suffix array for $T_1, T_2, \dots, T_m$ is associated with a suffix of some document T_i and the index i of the document the suffix comes from. We store these document indices in a separate array D, called *document array*, such that $D[i] = k$ if the i-th entry of the generalized suffix array for $T_1, T_2, \dots, T_m$ corresponds to a suffix coming from T_k. We also augment the document array D with a linear space data structure that answers queries $rank(k, i)$ (number of entries storing k before position i in D) and $select(k, i)$ (i-th entry from the left storing k). Using the result of [15], we can support such rank and select queries in $O(\log\log m)$ and $O(1)$ time respectively.

For each considered suffix array, we assume available, when needed, two auxiliary arrays: an inverse suffix array and another array, called the LCP-array, of longest common prefixes between each suffix and the preceding one in the lexicographic order. Moreover, we maintain a data structure that answers range minima queries (RMQ) on the LCP-array: for any $1 \le r_1 \le r_2 \le n$, find the minimum among $LCP[r_1], LCP[r_1+1], \dots, LCP[r_2]$. There

exists a linear space RMQ data structure that supports queries in constant time, see e.g., [4]. An RMQ query on the LCP-array computes the length of the longest common prefix of the suffixes of ranks r_1 and r_2, denoted $LCP(r_1, r_2)$.

2.2. Weighted Ancestor Problem

The *weighted ancestor* problem, defined in [9], is a generalization of the level ancestor problem [6, 5] for the case when tree edges are assigned positive weights. In this section, we present an improved solution to this problem that will be used later in the paper.

Consider a rooted tree $\mathcal{T}$ whose edges are assigned positive integer weights. For a node w, let $weight(w)$ denote the total weight of the edges on the path from the root to w; $depth(w)$ denotes the usual tree depth of w.

A weighted ancestor query $wa(v, q)$ asks, given a node v and a positive integer q, for the ancestor w of v of minimal depth such that $weight(w) \geq q$ ($wa(v, q)$ is undefined if there is no such node w).

Three previously known solutions [9, 20, 2] for the weighted ancestor problem achieve $O(\log \log W)$ query time using linear space, where W is the total weight of all tree edges. Our data structure also uses $O(n)$ space, but achieves a faster query time in many special cases. We prove the following result.

Theorem 1. *There exists an $O(n)$-space data structure that answers a weighted ancestor query $wa(v, q)$ in $O(\min(\sqrt{\log g / \log \log g}, \log \log q))$ time, where $g = \min(depth(wa(v, q)), depth(v) - depth(wa(v, q)))$.*

If every internal node is a branching node, we obtain the following corollary.

Corollary 1. *Suppose that every internal node in $\mathcal{T}$ has at least two children. There exists an $O(n)$-space data structure that finds $u = wa(v, q)$ in $O(\min(\sqrt{\log n_u / \log \log n_u}, \log \log q))$ time, where n_u is the number of leaves in the subtree of u.*

Our approach combines the heavy path decomposition technique of [2] with efficient data structures for finger searching in a set of integers.

Predecessor Queries. We will need the following result about a data structure that supports predecessor and successor queries. The predecessor of an integer q in a set S is $pred(q, S) = \max\{\, e \in S \mid e \leq q \,\}$. The successor of q in S is $succ(q, S) = \min\{\, e \in S \mid e \geq q \,\}$.

Lemma 1. *Let S be a set of n positive integers drawn from the universe of size U. Using an $O(n)$-space data structure, we can find $pred(q, S)$ and $succ(q, S)$ for any q in $O(\log\log q)$ time.*

PROOF. The set S is divided into consecutive groups S_i, $i = 1, 2, \ldots, \lceil n/d \rceil$, where $d = \log U$. Each S_i, except of the last one, contains d elements. For any $e \in S_i$ and $e' \in S_j$ iff $i < j$ then $e < e'$. All elements of S_i, $1 \leq i \leq \lceil n/\log U \rceil$, are stored in a data structure that supports predecessor and successor queries in $O(1)$ time [12].

A set S' contains the smallest element from each S_i. We store all elements of S' in a predecessor data structure D' described in [7]. By Theorem 3 from [7], there exists a data structure that uses $O(m \log U)$ space and answers predecessor queries on a set of m elements in $O(\log\log \Delta)$ time, where $\Delta \leq q$ is the distance between q and $pred(q, S)$. Hence, D' uses $O(n)$ space and enables us to find $pred(q, S')$ for any q in $O(\log\log q)$ time.

To find $pred(q, S)$, we identify $e' = pred(q, S')$. Let S_t be the group that contains e'. Obviously $pred(q, S_t)$ is the predecessor of q in S. We find $pred(q, S')$ in $O(\log\log q)$ time; we find $pred(q, S_t)$ in $O(1)$ time. Hence, a query $pred(q, S)$ is answered in $O(\log\log q)$ time.

If $e = pred(q, S)$ is known, then $e' = succ(q, S)$ is the element that follows e in the sorted list of elements. Hence, $succ(q, S)$ can also be found in $O(\log\log q)$ time.

Heavy Path Decomposition. A path π in $\mathcal{T}$ is heavy if every node u on π has at most twice as many nodes in its subtree as its child v on π. A tree $\mathcal{T}$ can be decomposed into paths using the following procedure: we find the longest heavy path π_r that starts at the root of $\mathcal{T}$ and remove all edges of π_r from $\mathcal{T}$. All remaining vertices of $\mathcal{T}$ belong to a forest; we recursively repeat the same procedure for every tree of that forest.

We can represent the decomposition into heavy paths using a tree $\mathbb{T}$. Each node $\mathbb{v}_j$ in $\mathbb{T}$ corresponds to a heavy path π_j in $\mathcal{T}$. A node $\mathbb{v}_j$ is a child of a node $\mathbb{v}_i$ in $\mathbb{T}$ if the head of π_j (i.e., the highest node in π_j) is a child of some node $u \in \pi_i$. In this case, some node of π_i has at least twice as many descendants as each node in π_j; hence, $\mathbb{T}$ has height $O(\log n)$.

$O(n \log n)$-Space Solution. Let $\mathbb{p}_j$ denote a root-to-leaf path in $\mathbb{T}$. For a node $\mathbb{v}$ in $\mathbb{T}$ let $weight(\mathbb{v})$ denote the weight of the head of π, where π is the heavy path represented by $\mathbb{v}$ in $\mathbb{T}$. We store a data structure $D(\mathbb{p}_j)$ that contains the values of $weight(\mathbb{v})$ for all nodes $\mathbb{v} \in \mathbb{p}_j$. $D(\mathbb{p}_j)$ contains $O(\log n)$ elements; hence, we can find the highest node $\mathbb{v} \in \mathbb{p}_j$ such that

$weight(\mathbb{v}) \geq q$ in $O(1)$ time. This can be achieved by storing the weights of all nodes from $\mathbb{p}_j$ in a data structure [12].

For every heavy path π in $\mathcal{T}$, we store the data structure $E(\pi)$ from [3] that contains the weights of all nodes $u \in \pi$ and supports the following queries: for an integer q, find the lightest node $u \in \pi$ such that $weight(u) \geq q$. Using Theorem 1.5 in [3], we can find such a node $u \in \pi$ in $O(\sqrt{\log n'/\log\log n'})$ time where $n' = \min(n_h, n_l)$, $n_h = |\{\, v \in \pi \,|\, weight(v) > weight(u) \,\}|$, and $n_l = |\{\, v \in \pi \,|\, weight(v) < weight(u) \,\}|$. Moreover, by Lemma 1, we can find the node u in $O(\log\log q)$ time. Thus $E(\pi)$ can be modified to support queries in $O(\min(\sqrt{\log n'/\log\log n'}, \log\log q))$ time.

For each node $u \in \mathcal{T}$ we store a pointer to the heavy path π that contains u and to the corresponding node $\mathbb{v} \in \mathbb{T}$.

A query $wa(v, q)$ can be answered as follows. Let $\mathbb{v}$ denote the node in $\mathbb{T}$ that corresponds to the heavy path containing v. Let $\mathbb{p}_j$ be an arbitrary root-to-leaf path in $\mathbb{T}$ that also contains $\mathbb{v}$. Using $D(\mathbb{p}_j)$ we can find the highest node $\mathbb{u} \in \mathbb{p}_j$, such that $weight(\mathbb{u}) \geq q$ in $O(1)$ time. Let π_t denote the heavy path in $\mathcal{T}$ that corresponds to the parent of $\mathbb{u}$, and π_s denote the path that corresponds to $\mathbb{u}$. If the weighted ancestor $wa(v, q)$ is not the head of π_s, then $wa(v, q)$ belongs to the path π_t. Using $E(\pi_t)$, we can find $u = wa(v, q)$ in $O(\min(\sqrt{\log n'/\log\log n'}, \log\log q))$ time where $n' = \min(n_h, n_l)$, $n_h = |\{\, v \in \pi_t \,|\, weight(v) > weight(u) \,\}|$, and $n_l = |\{\, v \in \pi_t \,|\, weight(v) < weight(u) \,\}|$.

All data structures $E(\pi_i)$ use linear space. Since there are $O(n)$ leaves in $\mathbb{T}$ and each path $\mathbb{p}_i$ contains $O(\log n)$ nodes, all $D(\mathbb{p}_i)$ use $O(n \log n)$ space.

Lemma 2. *There exists an $O(n \log n)$ space data structure that finds the weighted ancestor u in $O(\min(\sqrt{\log n'/\log\log n'}, \log\log q))$ time.*

$O(n)$-Space Solution. We can reduce the space from $O(n \log n)$ to $O(n)$ using a micro-macro tree decomposition. Let $\mathcal{T}_0$ be a tree induced by the nodes of $\mathcal{T}$ that have at least $\log n/8$ descendants. The tree $\mathcal{T}_0$ has at most $O(n/\log n)$ leaves. We construct the data structure described above for $\mathcal{T}_0$; since $\mathcal{T}_0$ has $O(n/\log n)$ leaves, its heavy path tree $\mathbb{T}_0$ also has $O(n/\log n)$ leaves. Therefore all structures $D(\mathbb{p}_j)$ use $O(n)$ words of space. All $E(\pi_i)$ also use $O(n)$ words of space. If we remove all nodes of $\mathcal{T}_0$ from $\mathcal{T}$, the remaining forest $\mathcal{F}$ consists of $O(n)$ nodes. Every tree $\mathcal{T}_i$, $i \geq 1$, in $\mathcal{F}$ consists of $O(\log n)$ nodes. Nodes of $\mathcal{T}_i$ are stored in a data structure that uses linear space and answers weighted ancestor queries in $O(1)$ time. This data structure will be described later in this section.

Suppose that a weighted ancestor $wa(v, q)$ should be found. If $v \in \mathcal{T}_0$, we answer the query using the data structure for $\mathcal{T}_0$. If v belongs to some $\mathcal{T}_i$ for $i \geq 1$, we check the weight w_r of $root(\mathcal{T}_i)$. If $w_r \leq q$, we search for $wa(v, q)$ in $\mathcal{T}_i$. Otherwise we identify the parent v_1 of $root(\mathcal{T}_i)$ and find $wa(v_1, q)$ in $\mathcal{T}_0$. If $wa(v_1, q)$ in $\mathcal{T}_0$ is undefined, then $wa(v, q) = root(\mathcal{T}_i)$.

Data Structure for a Small Tree. It remains to describe the data structure for a tree $\mathcal{T}_i$, $i \geq 1$. Since $\mathcal{T}_i$ contains a small number of nodes, we can answer weighted ancestor queries on $\mathcal{T}_i$ using a look-up table V. V contains information about any tree with up to $\log n/8$ nodes, such that node weights are positive integers bounded by $\log n/8$. For any such tree $\widetilde{\mathcal{T}}$, for any node v of $\widetilde{\mathcal{T}}$, and for any integer $q \in [1, \log n/8]$, we store the pointer to $wa(v, q)$ in $\widetilde{\mathcal{T}}$. There are $O(2^{\log n/4})$ different trees $\widetilde{\mathcal{T}}$ (see e.g., [5] for a simple proof); for any $\widetilde{\mathcal{T}}$, we can assign weights to nodes in less than $(\log n/8)!$ ways. For any weighted tree $\widetilde{\mathcal{T}}$ there are at most $(\log n)^2/64$ different pairs v, q. Hence, the table V contains $O(2^{\log n/4}(\log n)^2(\log n/8)!) = o(n)$ entries. We need only one look-up table V for all small trees $\mathcal{T}_i$.

We can now answer a weighted ancestor query on $\mathcal{T}_i$ using reduction to rank space [13]. The *rank* of a node u in a tree $\mathcal{T}$ is defined as $rank(u, \mathcal{T}) = |\{\, v \in \mathcal{T} \mid weight(v) \leq weight(u) \,\}|$. The successor of an integer q in a tree $\mathcal{T}$ is the lightest node $u \in \mathcal{T}$ such that $weight(u) \geq q$. The rank $rank(q, \mathcal{T})$ of an integer q is defined as the rank of its successor. Let $rank(\mathcal{T})$ denote the tree $\mathcal{T}$ in which the weight of every node is replaced with its rank. The weight of a node $u \in \mathcal{T}$ is not smaller than q if an only if $rank(u, \mathcal{T}) \geq rank(q, \mathcal{T})$. Therefore we can find $wa(v, q)$ in a small tree $\mathcal{T}_i$, $i \geq 1$, as follows. For every $\mathcal{T}_i$ we store a pointer to $\widetilde{\mathcal{T}}_i = rank(\mathcal{T}_i)$. Given a query $wa(v, q)$, we find $rank(q, \mathcal{T}_i)$ in $O(1)$ time using a q-heap [12]. Let v' be the node in $\widetilde{\mathcal{T}}_i$ that corresponds to the node v. We find $u' = wa(v', rank(q, \mathcal{T}_i))$ in $\widetilde{\mathcal{T}}_i$ using the table V. Then the node u in $\mathcal{T}_i$ that corresponds to u' is the weighted ancestor of v.

3. Cross-document Pattern Counting and Reporting

3.1. Counting

In this section we consider the problem of counting occurrences of a pattern $T_k[i..j]$ in a document T_ℓ.

Our data structure consists of the generalized suffix array GSA for documents $T_1, \ldots, T_m$ and individual suffix trees $\mathcal{T}_i$ for every document T_i.

For every suffix tree $\mathcal{T}_\ell$ we store a data structure of Theorem 1 supporting weighted ancestor queries on $\mathcal{T}_\ell$.

Our counting algorithm consists of two steps:

1. Using GSA and the corresponding document array D, we identify a position p of T_ℓ at which the query pattern $T_k[i..j]$ occurs, or determine that no such p exists,
2. Find the locus u of $T_k[i..j]$ in the suffix tree $\mathcal{T}_\ell$ using a weighted ancestor query and retrieve the number of leaves in the subtree rooted at u.

Let r be the position of $T_k[i..]$ in GSA. We find indexes $r_1 = select(\ell, rank(r, \ell))$ and $r_2 = select(\ell, rank(r, \ell) + 1)$ in $O(\log\log m)$ time (see Section 2.1). $GSA[r_1]$ (resp. $GSA[r_2]$) is the closest suffix from document T_ℓ that precedes (resp. follows) $T_k[i..]$ in the lexicographic order of suffixes. Observe now that $T_k[i..j]$ occurs in T_ℓ if and only if either $LCP(r_1, r)$ or $LCP(r, r_2)$ (or both) is no less than $j - i + 1$. If this holds, then the starting position p of $GSA[r_1]$ (respectively, starting position of $GSA[r_2]$) is the position of $T_k[i..j]$ in T_ℓ. Once such a position p is found, we jump to the leaf v of $\mathcal{T}_\ell$ that contains the suffix $T_\ell[p..]$.

The weighted ancestor $u = wa(v, (j - i + 1))$ is the locus of $T_k[i..j]$ in $\mathcal{T}_\ell$. This is because $T_\ell[p..p + j - i] = T_k[i..j]$. By Theorem 1, we can find u in $O(\log\log(j - i + 1))$ time.

Summing up, we obtain the following theorem.

Theorem 2. *For any $1 \le k, \ell \le m$ and $1 \le i \le j \le |T_k|$, we can count the number of occurrences of $T_k[i..j]$ in T_ℓ in $O(t + \log\log m)$ time, where $t = \min(\sqrt{\log occ / \log\log occ}, \log\log(j - i + 1))$ and occ is the number of occurrences. The underlying indexing structure takes $O(n)$ space and can be constructed in $O(n)$ time.*

3.2. Reporting

A reporting query asks for all occurrences of a substring $T_k[i..j]$ in T_ℓ.

To support reporting queries, we use the same two-step procedure as in Section 3.1, but make a slight change in the data structures: at Step 2, instead of using individual suffix trees for each of the documents, we use suffix arrays. The rest of the data structures is unchanged.

We first find an occurrence of $T_k[i..j]$ in T_ℓ (if there is one) using Step 1 of Section 3.1. Let p be the position of this occurrence in T_ℓ. We then jump to the corresponding entry r of the suffix array SA_ℓ for the document T_ℓ.

Let LCP_ℓ be the LCP-array of SA_ℓ. Starting with entry r, we visit adjacent entries t of SA_ℓ moving both to the left and to the right as long as $LCP_\ell[t] \geq j - i + 1$. While this holds, we report $SA_\ell[t]$ as an occurrence of $T_k[i..j]$. Note that the length of the longest common prefix of $SA_\ell[t]$ and $SA_\ell[p]$ is at least $j - i + 1$, as it is equal to the minimal value of $LCP_\ell[s]$, where s is between t and p. Hence, $SA_\ell[t]$ starts with $T_k[i..j]$, and $SA_\ell[t]$ is indeed an occurrence of $T_k[i..j]$.

No occurrence will be missed by the algorithm: if $SA_\ell[t]$ is an occurrence of $T_k[i..j]$, then the length of the longest common prefix of $SA_\ell[t]$ and $SA_\ell[p]$ is at least $|T_k[i..j]| = j - i + 1$ and therefore $LCP_\ell[s] \geq j - i + 1$ for any s between t and p. As a result, we obtain the following theorem.

Theorem 3. *All the occurrences of $T_k[i..j]$ in T_ℓ can be reported in $O(\log\log m + occ)$ time, where occ is the number of occurrences. The underlying indexing structure takes $O(n)$ space and can be constructed in $O(n)$ time.*

4. Variants of the Problem

4.1. Dynamic Counting and Reporting

In this section we focus on a *dynamic version* of counting and reporting problems, where dynamic operations are either addition or deletion of a document to the database. We assume that the total length of documents in the database is equal to n.

Recall that in the static case, counting occurrences of $T_k[i..j]$ in T_ℓ is done through the following two steps (Section 3.1): compute position p of some occurrence of $T_k[i..j]$ in T_ℓ, and find the locus of $T_\ell[p..p + j - i]$ in the suffix tree of T_ℓ.

For reporting queries (Section 3.2), Step 1 is unchanged, while Step 2 uses an individual suffix array for T_ℓ.

In the dynamic framework, we follow the same general two-step scenario. Note first that since Step 2, for both counting and reporting, uses data structures for individual documents only, it trivially applies to the dynamic case without changes. However, Step 1 requires serious modifications that we describe below.

Data Structures. Let L be a dynamic list of the suffixes of $T_1, T_2, \ldots, T_m$ in the lexicographic order. We will maintain the data structure for *Predecessor search on Dynamic Subsets of an Ordered Dynamic List problem* (*POLP* for short) of [18] on L. This data structure supports the following variant

of predecessor queries in $O(\log\log n)$ time: let $P_1, \ldots, P_m$ denote dynamic subsets of a list L; for any element e of L and for any i, $1 \leq i \leq m$, find the predecessor of e in P_i. In our setting, a subset P_k, $1 \leq k \leq m$, contains suffixes of document T_k.

We will also maintain the data structure of [11] (for a full version see [1]) on the suffixes of $T_1, T_2, \ldots, T_m$. This data structure allows us to compute the longest common prefix of any two suffixes in $O(1)$ time.

Queries. To identify an occurrence position of $T_k[i..j]$ in T_ℓ, we have to examine the two suffixes of T_ℓ closest to $T_k[i..]$ in L, respectively from right and from left. These suffixes correspond to the predecessor and the successor of $T_k[i..]$ in subset P_ℓ and can be found in $O(\log\log n)$ time by two queries to the $POLP$ data structure.

We then check if at least one of these two suffixes corresponds to an occurrence of $T_k[i..j]$ in T_ℓ. Similarly to Section 3, it is sufficient to compute the longest common prefix between each of these two suffixes and $T_k[i..]$, and compare these values with $(j - i + 1)$, which can be done in $O(1)$ time using the data structure of [11].

The query time bounds are summarized in the following lemma.

Lemma 3. *Using the above data structures, counting and reporting all occurrences of $T_k[i..j]$ in T_ℓ can be done respectively in time $O(\log\log n)$ and time $O(\log\log n + occ)$, where occ is the number of reported occurrences.*

Updates. We now explain how the data structures are updated. Suppose that we add a new document T. First we construct individual data structures for T in $O(|T|)$ time. Suffixes of T can be added to the $POLP$ data structure and the data structure [11] in $O(\log\log n + \log\log\sigma)$ expected time per suffix as described in [18].

Suppose now that we want to delete a document T. We start by deleting all individual data structures for T in $O(1)$ time per character. Then we have to update the $POLP$ data structure and the data structure [11]. Deletion of a suffix from the $POLP$ data structure takes $O(\log\log n)$ expected time [4]. To delete a suffix from the data structure of [11] one needs $O(1)$ time.

The following theorem summarizes the results of this section.

[4] In the paper of Kopelowitz [18], only insertions into the $POLP$ data structure are described. However, it is possible to modify this result, so that deletions are supported as well – see details at the full version in [19].

Theorem 4. *In the case when documents can be added or deleted dynamically, the number of occurrences of $T_k[i..j]$ in T_ℓ can be computed in time $O(\log\log n)$ and reporting these occurrences can be done in time $O(\log\log n + occ)$, where occ is their number and n is the total length of the documents in the database. The underlying data structure occupies $O(n)$ space and addition or deletion of a new document T takes $O(|T| \cdot (\log\log n + \log\log \sigma))$ expected time.*

4.2. Document Counting and Reporting

Consider a static collection of documents $T_1, \ldots, T_m$. In this section we focus on document reporting and counting queries: report or count the documents which contain at least one occurrence of $T_k[i..j]$, for some $1 \leq k \leq m$ and $i \leq j$.

For both counting and reporting, we use the generalized suffix tree, the generalized suffix array and the document array D for $T_1, T_2, \ldots, T_m$. We first retrieve the leaf of the generalized suffix tree labeled by $T_k[i..]$ and compute its highest ancestor u of string depth at least $j - i + 1$, using the weighted ancestor technique of Section 2.2. The suffixes of $T_1, T_2, \ldots, T_m$ starting with $T_k[i..j]$ (i.e. occurrences of $T_k[i..j]$) correspond then to the leaves of the subtree rooted at u, and vice versa. As shown in Section 3.1, this step takes $O(t)$ time, where $t = \min(\sqrt{\log occ / \log\log occ}, \log\log(j - i + 1))$ and occ is the number of occurrences of $T_k[i..j]$ (this time in all documents).

Once u has been computed, we retrieve the interval $[left(u)..right(u)]$ of ranks of all the leaves under interest. We are then left with the problem of counting/reporting distinct values in $D[left(u)..right(u)]$. This problem is exactly the same as the color counting/ color reporting problem that has been studied extensively (see e.g., [14] and references therein).

For color reporting queries, we can use the solution of [22] based on an $O(n)$-space data structure for RMQ, applied to (a transform of) the document array D. The pre-processing time is $O(n)$. Each document is then reported in $O(1)$ time, i.e. all relevant documents are reported in $O(ndocs)$ time, where $ndocs$ is their number. The whole reporting query then takes time $O(t + ndocs)$ for t defined above.

For counting, we use the solution described in [8]. The data structure requires $O(n)$ space and a color counting query takes $O(\log n)$ time. The following theorem presents a summary.

Theorem 5. *We can store a collection of documents $T_1, \ldots, T_m$ in a linear space data structure, so that for any pattern $P = T_k[i..j]$ all documents that contain P can be reported and counted in $O(t + ndocs)$ and $O(\log n)$ time*

respectively. Here $t = \min(\sqrt{\log occ / \log\log occ}, \log\log |P|)$*, ndocs is the number of documents that contain* P *and occ is the number of occurrences of* P *in all documents.*

4.3. Compact Counting, Reporting and Document Reporting

In this section, we show how our reporting and counting problems can be solved on *succinct* data structures [23].

Reporting and Counting. Our compact solution is based on compressed suffix arrays proposed by Grossi and Vitter [17]. A compressed suffix array for a text T uses $|CSA|$ bits of space and enables us to retrieve the position of the suffix of rank r, the rank of a suffix $T[i..]$, and the character $T[i]$ in time $Lookup(n)$. Different trade-offs between space usage and query time, $Lookup(n)$, can be achieved. E.g., the compressed suffix array described in [17, 25] uses $|CSA| = O((1 + 1/\varepsilon)n)$ and achieves $Lookup(n) = \log^{\varepsilon} n$ for any constant $\varepsilon > 0$ provided that a text T is over an alphabet of constant size. We refer the reader to [23] for an extensive survey of previous results.

Our data structure consists of a compressed generalized suffix array CSA for $T_1, \ldots, T_m$ and compressed suffix arrays CSA_i for each document T_i. In [26] it was shown that using $O(n)$ extra bits, the length of the longest common prefix of any two suffixes can be computed in $O(Lookup(n))$ time. Besides, the ranks of any two suffixes $T_k[s..]$ and $T_\ell[p..]$ can be compared in $O(Lookup(n))$ time: it suffices to compare $T_\ell[p + f]$ with $T_k[s + f]$ for $f = LCP(T_k[s..], T_\ell[p..])$.

Note that ranks of the suffixes of T_ℓ starting with $T_k[i..j]$ form an interval $[r_1..r_2]$. We use binary search on the compressed suffix array of T_ℓ to find r_1 and r_2. At each step of binary search, we compare a suffix of T_ℓ with $T_k[i..]$. Therefore $[r_1..r_2]$ can be found in $O(Lookup(n) \cdot \log n)$ time. Obviously, the number of occurrences of $T_k[i..j]$ in T_ℓ is $r_2 - r_1$. To report the occurrences, we compute the suffixes of T_ℓ with ranks in interval $[r_1..r_2]$.

Theorem 6. *All occurrences of* $T_k[i..j]$ *in* T_ℓ *can be counted in* $O(Lookup(n) \cdot \log n)$ *time and reported in* $O(Lookup(n) \cdot (\log n + occ))$ *time, where occ is the number of those. The underlying indexing structure takes* $2|CSA| + O(n)$ *bits of memory.*

Weighted Ancestor Problem on Compressed Suffix Trees. In order to adapt our document reporting solution (Section 4.2) to succinct data structures, we need first to adapt the solution of weighted ancestor problem accordingly. The following lemma provides a general time bound for weighted

ancestor queries depending on bounds for suffix tree operations provided by the data structure. Subsequently, we will use this Lemma to obtain an efficient succinct solution for the document reporting problem. In the rest of this section, we assume a constant-size alphabet.

Lemma 4. *Suppose we have a compressed suffix tree for a string of length n which supports the following operations:*

(i) Computing the number of leaves in the subtree of a node v in $Count(n)$ time,

(ii) Computing the string depth of a node v in $SDepth(n)$ time,

(iii) Computing the tree depth of a node v in $TDepth(n)$ time,

(iv) Computing the ancestor of v at level ℓ in $LAQ_T(n)$ time.

For any $\tau \geq 2$, we can add $O(n/\log^{\tau-2} n)$ bits to the compressed suffix tree and support weighted ancestor queries in time $O((SDepth(n) + LAQ_T(n) + Count(n))\tau \log\log n + TDepth(n))$, where weight of a node is defined to be its string depth.

Proof. Let $\mathcal{T}_0$ be a tree induced by the nodes of the suffix tree with at least $\log^\tau n$ leaves in their subtrees. The tree $\mathcal{T}_0$ has at most $O(n/\log^\tau n)$ leaves. We maintain a modified data structure of Section 2.2 for $\mathcal{T}_0$. Data structures $D(\mathbb{p}_j)$ for $\mathcal{T}_0$ are implemented exactly as in Section 2.2. Since $\mathcal{T}_0$ has $O(n/\log^\tau n)$ leaves, there are $O(n/\log^\tau n)$ data structures $D(\mathbb{p}_j)$. Each $D(\mathbb{p}_j)$ contains $O(\log n)$ elements of $\log n$ bits each. Therefore, all $D(\mathbb{p}_j)$ use $O(n/\log^{\tau-2} n)$ bits. We implement data structures $E(\pi_i)$ using a van Emde Boas data structure [28, 29] so that searching is supported in $O(\log\log n)$ time; $E(\pi_i)$ contains every $(\log^\tau n)$-th node from the heavy path π_i, so that all $E(\pi_i)$ use $O(n/\log^{\tau-1} n)$ bits. If we remove all nodes of $\mathcal{T}_0$ from the suffix tree, the remaining forest will consist of trees $\mathcal{T}_i$, $i \geq 1$, of height at most $\log^\tau n$.

Consider a query $wa(v, q)$. We first compute the number of leaves in the subtree rooted at v. If v has at least $\log^\tau n$ leaf descendants, then v belongs to $\mathcal{T}_0$ and we use the data structure for $\mathcal{T}_0$ to compute $wa(v, q)$. Using $o(n)$ additional bits, we can find the leftmost leaf descendant v_l of v [21]. Then, we find the node v'_l, such that v'_l is an ancestor of v_l and a leaf in $\mathcal{T}_0$. This can be done by binary search among the $\log^\tau n$ lowest ancestors of v_l in $O((LAQ_T(n) + SDepth(n))\tau \log\log n + TDepth(n))$ time. Then, we identify the path $\mathbb{p}_j$ from v'_l to the root of $\mathcal{T}_0$ and use $D(\mathbb{p}_j)$ to find the heavy path

π_i the node $wa(v,q)$ belongs to. Let u denote the lowest node in $E(\pi_i)$ with string depth at least q, and u' denote the parent of u in $E(\pi_i)$. The node $wa(v,q)$ belongs to the path from u to u'. Using binary search, we can find it in $O((LAQ_T(n) + SDepth(n))\tau \log\log n)$ time.

If v has less than $\log^\tau n$ leaf descendants, then v belongs to one of the micro-trees $\mathcal{T}_i$, $i \geq 1$. We first find the root r of $\mathcal{T}_i$ by binary search on the suffix tree in $O((LAQ_T(n) + Count(n))\tau \log\log n + TDepth(n))$ time. If its string depth is larger than q, then $wa(v,q) = wa(r,q)$ and we compute the latter as described before. Otherwise, we use binary search on the path from v to r and compute $wa(v,q)$ in $O((LAQ_T(n) + SDepth(n))\tau \log\log n)$ time.

As a specific case of Lemma 4, the compressed suffix tree of Sadakane [26] on top of the compressed suffix array of Grossi et al. [16] uses $(1 + 1/\varepsilon)nH_k + 6n + o(n)$ bits. For this data structure, $TDepth(n) = Count(n) = LAQ_T(n) = O(1)$ and $SDepth(n) = \log^\varepsilon n$. Combining these results with Lemma 4, we can compute $wa(v,q)$ in $O(\tau \log^\varepsilon n \log\log n)$ time using $(1+1/\varepsilon)nH_k + 6n + O(n/\log^{\tau-2} n) + o(n)$ bits of space, for any $\tau, \varepsilon > 0$. If $\tau = 3$, we have $O(\log^\varepsilon n \log\log n)$ query time and $(1+1/\varepsilon)nH_k + 6n + o(n)$ bits of space. This yields a competitive query time for weighted ancestor queries on the compressed suffix tree of Sadakane [26], see [24, 10] for comparison with previous works.

Document Reporting. As in Section 4.2 we use weighted ancestor queries on a generalized compressed suffix tree [26] to find the rank interval $[r_1..r_2]$ of suffixes that start with $T_k[i..j]$. That is, we first find a leaf of the tree representing $T_k[i..]$ in $O(\log^\varepsilon n)$ time and then we compute its highest ancestor u of string depth at least $j - i + 1$ in $O(\log^\varepsilon n \log\log n)$ time. Ranks r_1 and r_2 are ranks of the leftmost and rightmost leaves in the subtree of u, which can be computed in $O(1)$ time [26].

In [27] it was shown how to report, for any $1 \leq r_1 \leq r_2 \leq n$, all distinct documents T_f such that at least one suffix of T_f occurs at position r, $r_1 \leq r \leq r_2$, of the generalized suffix array. The construction uses $O(n + m \log \frac{n}{m})$ additional bits, and all relevant documents are reported in $O(\log^\varepsilon n \cdot ndocs)$ time, where *ndocs* is the number of documents that contain $T_k[i..j]$. Summing up, we obtain the following result.

Theorem 7. *All documents containing $T_k[i..j]$ can be reported in $O((\log\log n + ndocs)\log^\varepsilon n)$ time, where ndocs is the number of such documents and ε is an arbitrary positive constant. The underlying indexing structure takes $(1 + 1/\varepsilon)nH_k + O(n + m \log \frac{n}{m})$ bits of space.*

Acknowledgments. The authors gratefully acknowledge the help of Travis Gagie who suggested to use a compressed suffix tree in the succinct version of the document reporting problem. T.Starikovskaya has been supported by the mobility grant funded by the French Ministry of Foreign Affairs through the EGIDE agency, by RFBR grant 10-01-93109-CNRS-a, and by Dynasty Foundation.

References

[1] Amihood Amir, Gianni Franceschini, Roberto Grossi, Tsvi Kopelowitz, Moshe Lewenstein, and Noa Lewenstein. Managing unbounded-length keys in comparison-driven data structures with applications to on-line indexing. *CoRR*, abs/1306.0406, 2012.

[2] Amihood Amir, Gad M. Landau, Moshe Lewenstein, and Dina Sokol. Dynamic text and static pattern matching. *ACM Trans. Algorithms*, 3(2):19:1–19:24, 2007.

[3] Arne Andersson and Mikkel Thorup. Dynamic ordered sets with exponential search trees. *J. ACM*, 54(3):13:1–13:40, 2007.

[4] Michael A. Bender and Martin Farach-Colton. The LCA problem revisited. In *Proc. of the 4th Latin American Symposium on Theoretical Informatics*, volume 1776 of *Lecture Notes in Computer Science*, pages 88–94. Springer, 2000.

[5] Michael A. Bender and Martin Farach-Colton. The level ancestor problem simplified. *Theor. Comput. Sci.*, 321(1):5–12, 2004.

[6] Omer Berkman and Uzi Vishkin. Finding level-ancestors in trees. *J. Comput. Syst. Sci.*, 48(2):214–230, 1994.

[7] Prosenjit Bose, Karim Douïeb, Vida Dujmović, John Howat, and Pat Morin. Fast local searches and updates in bounded universes. *Comput. Geom. Theory Appl.*, 46(2):181–189, 2013.

[8] Panayiotis Bozanis, Nectarios Kitsios, Christos Makris, and Athanasios Tsakalidis. New upper bounds for generalized intersection searching problems. In *Proc. of the 22nd International Colloquium on Automata, Languages and Programming*, volume 944 of *Lecture Notes in Computer Science*, pages 464–474. Springer Berlin / Heidelberg, 1995.

[9] Martin Farach and S. Muthukrishnan. Perfect hashing for strings: Formalization and algorithms. In *Proc. of the 7th Annual Symposium on Combinatorial Pattern Matching*, volume 1075 of *Lecture Notes in Computer Science*, pages 130–140. Springer Berlin / Heidelberg, 1996.

[10] Johannes Fischer, Veli Mäkinen, and Gonzalo Navarro. Faster entropy-bounded compressed suffix trees. *Theor. Comput. Sci.*, 410(51):5354–5364, 2009.

[11] Gianni Franceschini and Roberto Grossi. A general technique for managing strings in comparison-driven data structures. In *Proc. of the 31st International Colloquium on Automata, Languages and Programming*, Lecture Notes in Computer Science, pages 606–617. Springer Berlin / Heidelberg, 2004.

[12] Michael L. Fredman and Dan E. Wilard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *J. Comput. Syst. Sci.*, 48(3):533–551, 1994.

[13] Harold N. Gabow, Jon Louis Bentley, and Robert Endre Tarjan. Scaling and related techniques for geometry problems. In *Proc. of the 16th ACM Symposium on Theory of Computing*, pages 135–143. ACM, 1984.

[14] Travis Gagie, Gonzalo Navarro, and Simon Puglisi. Colored range queries and document retrieval. In *Proc. of the 17th International Conference on String Processing and Information Retrieval*, volume 6393 of *Lecture Notes in Computer Science*, pages 67–81. Springer Berlin / Heidelberg, 2010.

[15] Alexander Golynski, J. Ian Munro, and S. Srinivasa Rao. Rank/select operations on large alphabets: a tool for text indexing. In *Proc. of the 17th ACM-SIAM Symposium on Discrete Algorithms*, pages 368–373. ACM Press, 2006.

[16] Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *Proc. of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 841–850, 2003.

[17] Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM J. Comput.*, 35(2):378–407, 2005; see also STOC 2000.

[18] Tsvi Kopelowitz. On-line indexing for general alphabets via predecessor queries on subsets of an ordered list. In *Proc. of the 53rd Annual IEEE Symposium on Foundations of Computer Science*, pages 283–292, 2012.

[19] Tsvi Kopelowitz. On-line indexing for general alphabets via predecessor queries on subsets of an ordered list. *CoRR*, abs/1208.3798, 2012.

[20] Tsvi Kopelowitz and Moshe Lewenstein. Dynamic weighted ancestors. In *Proc. of the 18th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 565–574, 2007.

[21] J. Ian Munro, Venkatesh Raman, and S. Srinivasa Rao. Space efficient suffix trees. *J. Algorithms*, 39(2):205–222, 2001.

[22] S. Muthukrishnan. Efficient algorithms for document retrieval problems. In *Proc. of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 657–666. SIAM, 2002.

[23] Gonzalo Navarro and Veli Mäkinen. Compressed full-text indexes. *ACM Comput. Surv.*, 39(1):2:1–2:61, 2007.

[24] Luís M. S. Russo, Gonzalo Navarro, and Arlindo L. Oliveira. Fully compressed suffix trees. *ACM Transactions on Algorithms*, 7(4):53:1–53:34, 2011.

[25] Kunihiko Sadakane. Compressed text databases with efficient query algorithms based on the compressed suffix array. In *Proc. of the 11th International Conference on Algorithms and Computation*, volume 1969 of *Lecture Notes in Computer Science*, pages 295–321. Springer Berlin / Heidelberg.

[26] Kunihiko Sadakane. Compressed suffix trees with full functionality. *Theor. Comp. Sys.*, 41(4):589–607, 2007.

[27] Kunihiko Sadakane. Succinct data structures for flexible text retrieval systems. *J. of Discrete Algorithms*, 5:12–22, 2007.

[28] Peter van Emde Boas, Robert Kaas, and Erik Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10:99–127, 1977.

[29] Dan E. Willard. Log-logarithmic worst-case range queries are possible in space Theta(n). *Information Processing Letters*, 17(2):81–84, 1983.