HAL
open science

Cross-document pattern matching

Gregory Kucherov, Yakov Nekrich, Tatiana Starikovskaya

To cite this version:

Gregory Kucherov, Yakov Nekrich, Tatiana Starikovskaya. Cross-document pattern matching. CPM 2012, Jul 2012, Helsinki, Finland. pp.196-207, 10.1007/978-3-642-31265-6_16 . hal-00789975

HAL Id: hal-00789975

https://hal.science/hal-00789975v1

Submitted on 19 Feb 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Cross-Document Pattern Matching

Gregory Kucherov[1], Yakov Nekrich[2], and Tatiana Starikovskaya[3,1]

[1] Laboratoire d'Informatique Gaspard Monge, Université Paris-Est & CNRS, Marne-la-Vallée, Paris, France, Gregory.Kucherov@univ-mlv.fr
[2] Department of Computer Science, University of Chile, Santiago, Chile, yakov.nekrich@googlemail.com
[3] Lomonosov Moscow State University, Moscow, Russia, tat.starikovskaya@gmail.com

Abstract. We study a new variant of the string matching problem called *cross-document string 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 level ancestor* problem.

1 Introduction

In this paper we study the following variant of the string matching problem that we call *cross-document string 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 $T_1, \ldots, T_m$, 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 string 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 level ancestor* problem [10, 1] 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 version where new documents can be added to 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 [15] 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 involved index data structure in compressed form.

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 $t = \min(\sqrt{\log \mathrm{occ}/\log\log \mathrm{occ}}, \log\log|P|)$, $P = T_k[i..j]$ 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 $O(\log\log m + \mathrm{occ})$,

(iii) in the dynamic case, when new documents can be dynamically added to the database, we are able to answer counting queries in time $O(\log n)$ and reporting queries in time $O(\log n + \mathrm{occ})$, whereas the updates take time $O(\log n)$ per character,

(iv) for the document counting and document reporting problems, our algorithms run in time $O(\log n)$ and $O(t + \mathrm{ndocs})$ 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 cross-document scenario (see Theorems 6 and 7 in Section 4.3).

For problems (i)-(iv), the involved data structures occupy $O(n)$ space. 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.

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 a 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 labelling the edges along the path from the root to v.

We will also use the 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 $T_1, \dots, T_m$ 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 $T_1, \dots, T_m$ corresponds to a suffix coming from T_k.

For each considered suffix array, we assume available, when needed, two auxiliary arrays: an inverted suffix array and another array, called the LCP-array, of longest common prefixes between each suffix and the preceding one in the lexicographic order.

2.2 Weighted Level Ancestor Problem

The *weighted level ancestor* problem, defined in [10], is a generalization of the level ancestor problem [6, 5] for the case when tree edges are assigned positive weights.

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

Two previously known solutions [10, 1] for weighted level ancestors 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 weighted ancestor query $\mathrm{wla}(v, q)$ in $O(\min(\sqrt{\log g / \log \log g}, \log \log q))$ time, where $g = \min(depth(\mathrm{wla}(v, q)), depth(v) - depth(\mathrm{wla}(v, q)))$.*

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 $w = \mathrm{wla}(v, q)$ in $O(\sqrt{\log n_w / \log \log n_w})$ time, where n_w is the number of leaves in the subtree of w.*

Our approach combines the heavy path decomposition technique of [1] with efficient data structures for finger searching in a set of integers. The proof is given in the Appendix.

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 level ancestor queries on $\mathcal{T}_\ell$. We also augment the document array D with an $O(n)$-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 [13], we can support such rank and select queries in $O(\log \log m)$ and $O(1)$ time respectively. Moreover, we construct a data structure that answers range minima queries (RMQ) on the LCP array: for any $1 \leq r_1 \leq r_2 \leq n$, find the minimum

among $LCP[r_1], \ldots LCP[r_2]$. 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 two suffixes $GSA[r_1]$ and $GSA[r_2]$, denoted $LCP(r_1, r_2)$.

Our counting algorithm consists of two stages. First, using GSA, 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. Then we find the locus of $T_k[i..j]$ in the suffix tree $\mathcal{T}_\ell$ using a weighted ancestor query.

Let r be the position of $T_k[i..]$ in the 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. $GSA[r_1]$ (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 level ancestor $u = \mathrm{wla}(v, (j - i + 1))$ 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 Corollary 1, we can find node u in $O(\sqrt{\log n_u / \log \log n_u})$ time, where n_u is the number of leaf descendants of u. Since u is the locus node of $T_k[i..j]$, n_u is the number of occurrences of $T_k[i..j]$ in T_ℓ. 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 $t = \min(\sqrt{\log \mathrm{occ} / \log \log \mathrm{occ}}, \log \log(j - i + 1))$ 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.

Compared to counting queries, we make a slight change in the data structures: instead of using suffix trees for individual documents T_i, 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) with the method described in 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 $T_k[i..j]$. It is easy to observe that the procedure is correct and that no occurrence is missing. 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 $O(\log\log m + \mathrm{occ})$ 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 the only dynamic operation consists in *adding a document to the database*[4].

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):

1. compute position p of some occurrence of $T_k[i..j]$ in T_ℓ,
2. in the suffix tree of T_ℓ, find the locus of string $T_\ell[p..p + j - i]$, and retrieve the number of leaves in the subtree rooted at u.

For reporting queries (Section 3.2), Step 1 is basically the same, while Step 2 is different and 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.

Since the suffix array is not well-suited for dynamic updates, at Step 1 we will use the generalized suffix tree for $T_1, T_2, \ldots, T_m$ hereafter denoted GST. For each suffix of $T_1, T_2, \ldots, T_m$ we store a pointer to the leaf of GST corresponding to this suffix.

We maintain a dynamic doubly-linked list EL corresponding to the Euler tour of the current GST. Each internal node of GST is stored in two copies in EL, corresponding respectively to the first and last visits

[4] document deletions are also possible to support but require some additional constructions that are left to the extended version of this paper

of the node during the Euler tour. Leaves of GST are kept in one copy. Observe that the leaves of GST appear in EL in the "left-to-right" order, although not consecutively.

On EL, we maintain the data structure of [3] which allows, given two list elements, to determine their order in the list in $O(1)$ time (see also [9]). Insertions of elements in the list are supported in $O(1)$ time too.

Furthermore, we maintain a balanced tree, denoted BT, whose leaves are elements of EL. Note that the size of EL is bounded by $2n$ (n is the size of GST) and the height of BT is $O(\log n)$. Since the leaves of GST are a subset of the leaves of BT, we call them *suffix leaves* to avoid the ambiguity.

Each internal node u of BT stores two kinds of information: (i) the rightmost and leftmost suffix leaves in the subtree of BT rooted at u, (ii) minimal LCP value among all suffix leaves in the subtree of BT rooted at u.

Finally, we will also need an individual suffix array for each inserted document T_i.

We are now in position to describe the algorithm of Step 1. Like in the static case, we first retrieve the leaf of GST corresponding to suffix $T_k[i..]$. To identify a position of an occurrence of $T_k[i..j]$ in T_ℓ, we have to examine the two closest elements in the list of leaves of GST, one from right and from left, corresponding to suffixes of T_ℓ. To find these two suffixes, we perform a binary search on the suffix array for T_ℓ using order queries of [3] on EL. This step takes $O(\log |T_\ell|)$ time.

We then check if at least one of these two suffixes corresponds to an occurrence of $T_k[i..j]$ in T_ℓ. In a similar way to Section 3, we have to compute the longest common prefix between each of these two suffixes and $T_k[i..]$, and compare this value with $(j-i+1)$. This amounts to computing the minimal LCP value among all the suffixes of the corresponding range.

This can be done in $O(\log n)$ time by using a standard range trees approach [**?**]: for any sublist of EL we can retrieve $O(\log n)$ nodes v_i that cover it. The least among all minimal LCP values stored in nodes v_i is the minimal LCP value for the specified range of suffixes.

The query time bounds are summarized in the following lemma.

Lemma 1. *Using the above data structures, counting and reporting all occurrences of $T_k[i..j]$ in T_ℓ can be done respectively in time $O(\log n)$ and time $O(\log n + \mathrm{occ})$, where* occ *is the number of reported occurrences.*

We now explain how the involved data structures are updated. Suppose that we add a new document T_{m+1}. Extending the generalized suffix

tree by T_{m+1} is done in time $O(|T_{m+1}|)$ by McCreight's or Ukkonen's algorithm, i.e. in $O(1)$ amortized time per symbol.

When a new node v is added to a suffix tree, the following updates should be done (in order):

(i) insert v at the right place of the list EL (in two copies if v is an internal node),
(ii) rebalance the tree BT if needed,
(iii) if v is a leaf of GST (i.e. a suffix leaf of BT), update LCP values and rightmost/leftmost suffix leaf information in BT,

To see how update (i) works, we have to recall how suffix tree is updated when a new document is inserted. Two possible updates are creation of a new internal node v by splitting an edge into two (edge subdivision) and creating a new leaf u as a child of an existing node. In the first case, we insert the first copy of v right after the first copy of its parent, and the second copy right before the second copy of its parent. In the second case, the parent of u has already at least one child, and we insert u either right after the second (or the only) copy of its left sibling, or right before the first (or the only) copy of its right sibling.

Rebalancing the tree BT (update (ii)) is done using standard methods. Observe that during the rebalancing we may have to adjust the LCP and rightmost/leftmost suffix leaf information for internal nodes, but this is easy to do as only a constant number of local modifications is done at each level.

Update (iii) is triggered when a new leaf u is created in GST and added to EL. First of all, we have to compute the LCP value for u and possibly to update the LCP value of the next suffix leaf u' to the right of u in EL. This is done in $O(1)$ time as follows. At the moment when u is created, we memorize the string depth of its parent $D = d(parent(u))$. Recall that the parent of u already has at least one child before u is created. If u is neither the leftmost nor the rightmost child of its parent, then we set $LCP(u) = D$ and $LCP(u')$ remains unchanged (actually it also equals D). If u is the leftmost child of its parent, then we set $LCP(u) = LCP(u')$ and then $LCP(u') = D$. Finally, if u is the rightmost child, then $LCP(u) = D$ and $LCP(u')$ remains unchanged.

We then have to follow the path in BT from the new leaf u to the root and possibly update the LCP and rightmost/leftmost suffix leaf information for all nodes on this path. These updates are straightforward. Furthermore, during this traversal we also identify suffix leaf u' (as the leftmost child of the first right sibling encountered during the traversal),

update its LCP value and, if necessary, the LCP values on the path from u' to the root of BT. All these steps take time $O(\log n)$.

Thus, updates of all involved data structures take $O(\log n)$ time per symbol. The following theorem summarizes the results of this section.

Theorem 4. *In the case when documents can be added dynamically, the number of occurrences of $T_k[i..j]$ in T_ℓ can be computed in time $O(\log n)$ and reporting these occurrences can be done in time $O(\log n + occ)$, where occ is their number. The underlying data structure occupies $O(n)$ space and an update takes $O(\log n)$ time per character.*

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, 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 labelled by $T_k[i..]$ and compute its highest ancestor u of string depth at least $j - i + 1$, using the weighted level 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 $t = \min(\sqrt{\log \mathrm{occ}/\log\log \mathrm{occ}}, \log\log(j - i + 1))$ 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., [12] and references therein).

For color reporting queries, we can use the solution of [15] 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 $O(\mathrm{ndocs})$ time, where ndocs is their number. The whole reporting query then takes time $O(t + \mathrm{ndocs})$ for t defined above.

For counting, we use the solution described in [7]. 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 $O(t + \mathrm{ndocs})$ and $O(\log n)$ time respectively. Here $t = \min(\sqrt{\log \mathrm{occ} / \log\log \mathrm{occ}}, \log\log |P|)$, 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 [16].

Reporting and Counting. Our compact solution is based on compressed suffix arrays [14]. 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 can be achieved (see [16] for a survey).

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 [17] 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 a binary search on the compressed suffix array of T_ℓ to find r_1 and r_2. At each step of the 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 $O((\log n + \mathrm{occ}) Lookup(n))$ time, where* occ *is the number of those. The underlying indexing structure takes $2|CSA| + O(n + m \log \frac{n}{m})$ bits of memory.*

Document Reporting Again, we use a binary search on the generalized suffix array to find the rank interval $[r_1..r_2]$ of suffixes that start with $T_k[i..j]$. This can be done in $O(Lookup(n) \cdot \log n)$ time.

In [18], 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 $O(Lookup(n) \cdot \text{ndocs})$ 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 $O((\log n + \text{ndocs})Lookup(n))$ time, where* ndocs *is the number of those. The underlying indexing structure takes $2|CSA| + O(n + m \log \frac{n}{m})$ bits of space.*

Acknowledgments: T.Starikovskaya has been supported by the mobility grant funded by the French Ministry of Foreign Affairs through the EGIDE agency and by a grant 10-01-93109-CNRS-a of the Russian Foundation for Basic Research. Part of this work has been done during a one-month stay of Y.Nekrich at the Marne-la-Vallée University supported by the BEZOUT grant of the French government. The authors also would like to thank Tsvi Kopelowitz for fruitful discussions and valuable comments.

References

1. A. Amir, G. M. Landau, M. Lewenstein, and D. Sokol. Dynamic text and static pattern matching. *ACM Trans. Algorithms*, 3, May 2007.
2. A. Andersson and M. Thorup. Dynamic ordered sets with exponential search trees. *J. ACM*, 54(3):13, 2007.
3. M. A. Bender, R. Cole, E. D. Demaine, M. Farach-Colton, and J. Zito. Two simplified algorithms for maintaining order in a list. In *Proceedings of the 10th Annual European Symposium on Algorithms*, ESA '02, pages 152–164, London, UK, UK, 2002. Springer-Verlag.
4. M. A. Bender and M. Farach-Colton. The lca problem revisited. In *Proceedings of the 4th Latin American Symposium on Theoretical Informatics*, LATIN '00, pages 88–94, London, UK, 2000. Springer-Verlag.
5. M. A. Bender and M. Farach-Colton. The level ancestor problem simplified. *Theor. Comput. Sci.*, 321(1):5–12, 2004.
6. O. Berkman and U. Vishkin. Finding level-ancestors in trees. *J. Comput. Syst. Sci.*, 48(2):214–230, 1994.
7. P. Bozanis, N. Kitsios, C. Makris, and A. K. Tsakalidis. New upper bounds for generalized intersection searching problems. In *Automata, Languages and Programming, 22nd International Colloquium, (ICALP) Proceedings*, pages 464–474, 1995.
8. G. S. Brodal, P. Davoodi, and S. S. Rao. Path minima queries in dynamic weighted trees. In *Proceedings of the 12th International Symposium on Algorithms and Data Structures*, WADS'11, pages 290–301, Berlin, Heidelberg, 2011. Springer-Verlag.

9. P. Dietz and D. Sleator. Two algorithms for maintaining order in a list. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*, STOC '87, pages 365–372, New York, NY, USA, 1987. ACM.
10. M. Farach and S. Muthukrishnan. Perfect hashing for strings: Formalization and algorithms. In *Proceedings of the 7th Annual Symposium on Combinatorial Pattern Matching*, CPM '96, pages 130–140, London, UK, 1996. Springer-Verlag.
11. M. L. Fredman and D. E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *J. Comput. Syst. Sci.*, 48(3):533–551, 1994.
12. T. Gagie, G. Navarro, and S. J. Puglisi. Colored range queries and document retrieval. In *Proceedings of the 17th International Conference on String Processing and Information Retrieval*, SPIRE'10, pages 67–81, Berlin, Heidelberg, 2010. Springer-Verlag.
13. A. Golynski, J. I. Munro, and S. S. Rao. Rank/select operations on large alphabets: a tool for text indexing. In *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2006, Miami, Florida, USA, January 22-26, 2006*, pages 368–373. ACM Press, 2006.
14. R. Grossi and J. S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching (extended abstract). In *Proceedings of the thirty-second annual ACM symposium on Theory of computing*, STOC '00, pages 397–406, New York, NY, USA, 2000. ACM.
15. S. Muthukrishnan. Efficient algorithms for document retrieval problems. In *Proc. of the 13th Annual ACM-SIAM Symp. on Discrete Algorithms, January 6-8, 2002, San Francisco, CA, USA*, pages 657–666. ACM/SIAM, 2002.
16. G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Comput. Surv.*, 39, April 2007.
17. K. Sadakane. Compressed suffix trees with full functionality. *Theory Comput. Syst.*, 41(4):589–607, 2007.
18. K. Sadakane. Succinct data structures for flexible text retrieval systems. *J. of Discrete Algorithms*, 5:12–22, March 2007.

Appendix: Proof of Theorem 1

Here we prove Theorem 1. We use the heavy path decomposition technique of [1].

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 in every tree of that forest.

We can represent the decomposition into heavy paths using a tree $\mathbb{T}$. Each node $\mathtt{v}_j$ in $\mathbb{T}$ corresponds to a heavy path π_j in $\mathcal{T}$. A node $\mathtt{v}_j$ is a child of a node $\mathtt{v}_i$ in $\mathbb{T}$ if the head of π_j (i.e., the highest node in π_j) is a child of some node $u \in \pi_i$. Some node in 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 the q-heap [11].

For every heavy path π_j, we store the data structure $E(\pi_j)$ from [2] that contains the weights of all nodes $u \in \pi_j$ and supports the following queries: for an integer q, find the lightest node $u \in \pi_j$ such that $weight(u) \geq q$. Using Theorem 1.5 in [2], we can find such a node $u \in \pi_j$ in $O(\sqrt{\log n' / \log\log n'})$ time where $n' = \min(n_h, n_l)$, $n_h = |\{\, v \in p_j \,|\, weight(v) > weight(u) \,\}|$, and $n_l = |\{\, v \in p_j \,|\, weight(v) < weight(u) \,\}|$. Moreover, we can also find the node u in $O(\log\log q)$ time; we will show how this can be done in the full version of this paper. Thus $E(\pi_j)$ 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 $\mathrm{wla}(v, q)$ 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 $\mathrm{wla}(v, q)$ is not the head of π_s, then $\mathrm{wla}(v, q)$ belongs to the path π_t. Using $E(\pi_t)$, we can find $u = \mathrm{wla}(v, q)$ 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 level 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 $\mathrm{wla}(v, q)$ 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 $\mathrm{wla}(v, q)$ in $\mathcal{T}_i$. Otherwise we identify the parent v_1 of $root(\mathcal{T}_i)$ and find $\mathrm{wla}(v_1, q)$ in $\mathcal{T}_0$. If $\mathrm{wla}(v_1, q)$ in $\mathcal{T}_0$ is undefined, then $\mathrm{wla}(v, q) = root(\mathcal{T}_i)$.

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 level 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 $\mathrm{wla}(v, q)$ 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 mini-trees $\mathcal{T}_i$.

We can now answer a weighted level ancestor query on $\mathcal{T}_i$ using reduction to rank space. The *rank* of a node u in a tree $\mathcal{T}$ is defined as $\mathrm{rank}(u, \mathcal{T}) = |\{\, v \in \mathcal{T} \mid weight(v) \leq weight(u) \,\}|$. 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 $\mathrm{rank}(q, \mathcal{T})$ of an integer q is defined as the rank of its successor. Let $\mathrm{rank}(\mathcal{T})$ 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 $\mathrm{rank}(u, \mathcal{T}) \geq \mathrm{rank}(q, \mathcal{T})$. Therefore we can find $\mathrm{wla}(v, q)$ in a small tree $\mathcal{T}_i$, $i \geq 1$, as follows. For every $\mathcal{T}_i$ we store a pointer to $\widetilde{\mathcal{T}_i} = \mathrm{rank}(\mathcal{T}_i)$. Given a query $\mathrm{wla}(v, q)$, we find $\mathrm{rank}(q, \mathcal{T}_i)$ in $O(1)$ time using a q-heap [11]. Let v' be the node in $\widetilde{\mathcal{T}_i}$ that corresponds to the node v. We find $u' = \mathrm{wla}(v', \mathrm{rank}(q, \mathcal{T}_i))$ in $\widetilde{\mathcal{T}_i}$ using the table V. Then the node u in $\mathcal{T}_i$ that corresponds to u' is the weighted level ancestor of v.