Journal of Discrete Algorithms 24 (2014) 40–47

Contents lists available at SciVerse ScienceDirect

Journal of Discrete Algorithms

www.elsevier.com/locate/jda

Cross-document pattern matching ☆

Tsvi Kopelowitz [a], Gregory Kucherov [b], Yakov Nekrich [c,1], Tatiana Starikovskaya [d,*]

[a] Weizmann Institute of Science, Rehovot, Israel
[b] Laboratoire d'Informatique Gaspard Monge, Université Paris-Est & CNRS, Marne-la-Vallée, Paris, France
[c] Department of Electrical Engineering & Computer Science, University of Kansas, Lawrence, USA
[d] Lomonosov Moscow State University, Moscow, Russia

ARTICLE INFO

Article history:
Available online 31 May 2013

Keywords:
Algorithms
Pattern matching
Document reporting
Weighted ancestor problem

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.

© 2013 Elsevier B.V. All rights reserved.

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.

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

* Corresponding author.
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).

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

1570-8667/$ – see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.jda.2013.05.002

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 $T[i]T[i+1]\ldots T[j]$ 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 $T_1, T_2, \ldots, T_m$ that can be viewed as the suffix tree for the string $T_1\$_1T_2\$_2\ldots T_m\$_m$. 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 $T_1, T_2, \ldots, T_m$. 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 $T_1, T_2, \ldots, T_m$ 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 $T_1, T_2, \ldots, T_m$ 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 $1 \leqslant r_1 \leqslant r_2 \leqslant n$, find the minimum among $LCP[r_1], LCP[r_1+1], \ldots, LCP[r_2]$. 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 the suffixes of ranks r_1 and r_2, denoted $LCP(r_1, r_2)$.

Download English Version:

https://daneshyari.com/en/article/430860

Download Persian Version:

https://daneshyari.com/article/430860

Daneshyari.com