Computing Longest Common Substrings Via Suffix Arrays

Maxim A. Babenko and Tatiana A. Starikovskaya

Moscow State University
max@adde.math.msu.su,
tat.starikovskaya@gmail.com

Abstract. Given a set of N strings $A = \{\alpha_1, \ldots, \alpha_N\}$ of total length n over alphabet Σ one may ask to find, for each $2 \leq K \leq N$, the longest substring β that appears in at least K strings in A. It is known that this problem can be solved in $O(n)$ time with the help of suffix trees. However, the resulting algorithm is rather complicated (in particular, it involves answering certain least common ancestor queries in $O(1)$ time). Also, its running time and memory consumption may depend on $|\Sigma|$.

This paper presents an alternative, remarkably simple approach to the above problem, which relies on the notion of suffix arrays. Once the suffix array of some auxiliary $O(n)$-length string is computed, one needs a simple $O(n)$-time postprocessing to find the requested longest substring. Since a number of efficient and simple linear-time algorithms for constructing suffix arrays has been recently developed (with constant not depending on $|\Sigma|$), our approach seems to be quite practical.

1 Introduction

Consider the following problem:

> (LCS) *Given a collection of N strings $A = \{\alpha_1, \ldots, \alpha_N\}$ over alphabet Σ find, for each $2 \leq K \leq N$, the longest string β that is a substring of at least K strings in A.*

It is known as a generalized version of the *Longest Common Substring* (LCS) problem and has a plenty of practical applications, see [Gus97] for a survey.

Even in the simplest case of $N = K = 2$ a linear-time algorithm is not easy. A standard approach is to construct the so-called *generalized suffix tree* T (see [Gus97]) for $\alpha_1\$_1$ and $\alpha_2\$_2$, which is a compacted symbol trie that captures all the substrings of $\alpha_1\$_1$, $\alpha_2\$_2$. Here $\$_i$ are special symbols (called *sentinels*) that are distinct and do not appear in α_1 and α_2. Then, nodes of T are examined in a bottom-up fashion and those having sentinels of both types in their subtrees are listed. Among these nodes of T let us choose a node v with the largest *string depth* (which is the length of the string obtained by reading letters along the path from root to v). The string that corresponds to v in T is the answer. See [Gus97] for more details.

E.A. Hirsch et al. (Eds.): CSR 2008, LNCS 5010, pp. 64–75, 2008.

In practice, the above approach is not very efficient since it involves computing T. Several linear-time algorithms for the latter task are known (possibly, the most famous one is due to Ukkonen [Ukk95]). However, suffix trees are still not very convenient. They do have linear space bound but the hidden constants can be pretty large. Most of modern algorithms for computing suffix trees have the time bound of $O(n \log |\Sigma|)$ (where n denotes the length of a string). Hence, their running time depends on $|\Sigma|$. Moreover, achieving this time bound requires using balanced search trees to store arcs. The latter data structures further increase constants in both time- and space-bounds making these algorithms rather impractical. Other options include employing hashtables or assuming that $|\Sigma|$ is small and using direct addressing to access arcs leaving each node. These approaches have their obvious disadvantages.

If one assumes that N and K are arbitrary then additional complications arise. Now we are interested in finding the deepest (in sense of string depth) node v in T such that the tree rooted at v contains sentinels of at least K distinct kinds. Moreover, this routine should run in parallel for all possible values of K and take linear time. This seems to be an involved task. A possible solution requires answering *Least Common Ancestor* (LCA) queries on T in $O(1)$ time, e.g. using a method from [BFC00]. Reader may refer to [Gus97] for a complete outline.

In this paper we present an alternative approach that is based on the notion of *suffix arrays*. The latter were introduced by Manber and Myers [MM90] in an attempt to overcome the issues that are inherent to suffix trees. The *suffix array* (SA) of string α having length n is merely an array of n integers that indicate the lexicographic order of non-empty suffixes of α (see Section 2 for a precise definition). Its simplicity and compactness make it an extremely useful tool in modern text processing. Originally, an $O(n \log n)$-time algorithm for constructing SA was suggested [MM90]. This algorithm is not very practical. Subsequently, much simpler and faster algorithms for computing SA were developed. We particularly mention an elegant approach of Kärkkäinen and Sanders [KS03]. A comprehensive practical evaluation of different algorithms for constructing SA is given in [PST07].

We present two algorithms. The first one, which is simpler, assumes that parameter K is fixed. It first builds an auxiliary string α by concatenating strings α_i and intermixing them with sentinels $\$_i$ $(1 \le i \le N)$ and then constructs the suffix array for string α. Also, an additional *LCP array* is constructed. Finally, a sliding window technique is applied to these arrays to obtain the answer. Altogether, the running time is linear and does not depend on $|\Sigma|$.

The second algorithm deals with all possible values of K simultaneously. Its initial stage stage is similar: string α, suffix array for α, and LCP array are constructed. Then, *Cartesian tree* (CT) is constructed from LCP array. Finally, a certain postprocessing aimed to count the number of distinct types of nodes appearing in subtrees of CT is used. It should be noticed that this postprocessing does not require answering any least common ancestor queries.

The paper is organized as follows. Section 2 gives a formal background, introduces useful notation and definitions. It also explains the notion of suffix arrays