Computing Longest Common Substrings Via Suffix Arrays

Maxim A. Babenko * Tatiana A. Starikovskaya **

Moscow State University

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 a fixed integer K, $2 \leq K \leq N$, 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. 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 and an integer K, $2 \leq K \leq N$, find 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.

* Email: max@adde.math.msu.su
** Email: tat.starikovskaya@gmail.com

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.

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].

The algorithm presented first builds an auxiliary string α by concatenating strings α_i and intermixing them with sentinels $\$_i$ $(1 \leq i \leq N)$ 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 paper is organized as follows. Section 2 gives a formal background, introduces useful notation and definitions. It also explains the notion of suffix arrays and indicates how an auxiliary LCP array is constructed in linear time. Section 3 presents the algorithm.

2 Preliminaries

We shall start with a number of definitions first. In what follows we assume that a finite non-empty set Σ (called an *alphabet*) is fixed. The elements of Σ are *letters* or *symbols*. A finite ordered sequence of letters (possibly empty) is called a *string*.

We assume the usual RAM model of computation [AUH74]. Letters are treated just as integers in range $\{1, \ldots, |\Sigma|\}$, so one can compare any pair of them in $O(1)$ time. This *lexicographic* order on Σ is linear and can be extended in a standard way to the set of strings in Σ. We write $\alpha < \beta$ to denote that α lexicographically precedes β; similarly for other relation signs.

We usually use Greek symbols to denote strings. Letters in a string are numbered starting from 1, that is, for a string α of *length* k its letters are $\alpha[1], \ldots, \alpha[k]$. The length k of α is denoted by $|\alpha|$. The *substring* of α from position i to position j (inclusively) is denoted by $\alpha[i..j]$. Also, if $i = 1$ or $j = |\alpha|$ then these indices are omitted from the notation and we write just $\alpha[..j]$ and $\alpha[i..]$. String $\beta = \alpha[..j]$ is called a *prefix* of α. Similarly, if $\beta = \alpha[i..]$ then β is called a *suffix* of α. For a set of strings S let $lcp(S)$ denote the longest common prefix of all strings in S.

Recall that A stands for the collection of the input strings α_i. We start with an almost trivial observation:

Proposition 1. *Let $B = \{\beta_1, \ldots, \beta_m\} \subseteq A$ be an arbitrary subset of A obeying $m \geq K$. Let γ_i be an arbitrary suffix of β_i for each $1 \leq i \leq m$. Solving* (LCS) *amounts to computing the longest string among*

$$lcp(\gamma_1, \ldots, \gamma_m)$$

where maximum is taken over all possible choices of subsets B and suffixes $\{\gamma_i\}$.

Let us combine the strings in A as follows:

$$\alpha = \alpha_1 \$_1 \alpha_2 \$_2 \ldots \alpha_N \$_N \tag{1}$$

Here $\$_i$ are pairwise distinct *sentinel* symbols not appearing in strings of A. We assume that these sentinels are lexicographically smaller than other (normal) symbols. The lexicographic order between sentinels is not important.

String α captures all needed information about set A. For each index i ($1 \leq i \leq N$) and a position j in α_i ($1 \leq j \leq |\alpha_i|$) one may consider the *corresponding* position $p(i, j)$ in α:

$$p(i, j) := \sum_{k=1}^{i-1} (|\alpha_k| + 1) + j$$

Positions in α of the form $p(i, j)$ are called *essential*; the remaining positions (those containing sentinels) are called *unessential*.

Let us employ the following metaphor: for each $1 \leq i \leq N$ and $1 \leq j \leq |\alpha_i|$ we say that position $p(i, j)$ is *of type i* (it corresponds to the i-th string). Remaining (unessential) positions k in α are said to be of type 0.

Now taking into account the properties of sentinels one can easily derive the following claim from Proposition 1:

Proposition 2. *Let $P = \{p_1, \ldots, p_m\}$ be an arbitrary set of essential positions in α such that elements of P are of at least K distinct types. Solving* (LCS) *amounts to computing the longest string among*

$$lcp\big(\alpha[p_1..], \ldots, \alpha[p_m..]\big)$$

where maximum is taken over all possible choices of P.

	suffixes	SA	sorted suffixes	lcp
1	`mississippi`	11	`i`	1
2	`ississippi`	8	`ippi`	1
3	`ssissippi`	5	`issippi`	4
4	`sissippi`	2	`ississippi`	0
5	`issippi`	1	`mississippi`	0
6	`ssippi`	10	`pi`	1
7	`sippi`	9	`ppi`	0
8	`ippi`	7	`sippi`	2
9	`ppi`	4	`sissippi`	1
10	`pi`	6	`ssippi`	3
11	`i`	3	`ssissippi`	

Fig. 1. String `mississippi`, its suffixes, and the corresponding suffix and LCP arrays.

This does not seem very promising at the first glance. However, the longest common prefix computation exhibits a nice structure when it is applied to suffixes of a fixed string (in our case, string α).

To explain this structure we first introduce the notion of suffix arrays. Let ω be an arbitrary string of length n. Consider its non-empty suffixes

$$\omega[1..],\ \omega[2..],\ \ldots,\ \omega[n..]$$

and order them lexicographically. Let $SA(i)$ denote the starting position of the suffix appearing on the i-th place ($1 \le i \le n$) in this order:

$$\omega[SA(1)..] < \omega[SA(2)..] < \ldots < \omega[SA(n)..]$$

Clearly, SA is determined uniquely since all suffixes of ω are distinct. An example is depicted in Fig. 1.

Since SA is a permutation of $\{1, \ldots, n\}$ there must be an inverse correspondence. We denote it by *rank*; that is, *rank* is also a permutation of $\{1, \ldots, n\}$ and

$$SA(rank(i)) = i \qquad \text{holds for all } 1 \le i \le n.$$

Historically, the first algorithm to compute SA was due to Manber and Myers [MM90]; this algorithm takes $O(n \log n)$ time. Currently, simple linear-time algorithms for this task are known, see [PST07] for a list. The latter linear time bounds do not depend on $|\Sigma|$.

However, knowing SA is not enough for our purposes. We also have to precompute the lengths of longest common prefixes for each pair of consequent suffixes (with respect to the order given by SA). More formally,

$$lcp(i) := \big|lcp\big(\omega[SA(i)..], \omega[SA(i+1)..]\big)\big| \qquad \text{for all } 1 \le i < n.$$

This gives rise to array lcp of length $n-1$; we call it the *LCP array of* ω. The latter array not only enables to answer LCP queries for consequent (w.r.t. SA) suffixes of ω but also carries enough information to answer *any* such query. Formally [MM90]:

Lemma 1. *For each pair* $1 \le i < j \le n$ *one has*

$$\big|lcp\big(\omega[SA(i)..], \omega[SA(j)..]\big)\big| = \min_{i \le k < j} lcp(k)$$

Knowing the suffix array, LCP array may be constructed in $O(n^2)$ time by a brute-force method. However, an elegant modification ([KLA+01], see also [CR03]) allows to compute the longest common prefix for a pair of consequent suffixes in $O(1)$ amortized time. The key is to compute these values in a particular order, namely

$$lcp(rank(1)),\ lcp(rank(2)),\ \ldots,\ lcp(rank(n))$$

The efficiency of this approach relies on the following fact [KLA+01]:

Lemma 2. $lcp(rank(i+1)) \ge lcp(rank(i)) - 1$ *for each* $1 \le i < n$ *such that* $rank(i) < n$ *and* $rank(i+1) < n$.

Hence, when computing the value of $lcp(rank(i+1))$ one can safely skip $lcp(rank(i)) - 1$ initial letters of $\omega[i+1..]$ and $\omega[SA(rank(i+1)+1)..]$. This easily implies the required linear time bound for the whole computation.

3 The algorithm

The algorithm works as follows. It first combines the input strings into string α of length L (see (1)) and invokes the suffix array computation algorithm thus obtaining the suffix array SA for α. It also constructs array lcp in $O(L)$ time as described in Section 2.

Refer to Proposition 2 and consider an arbitrary set of essential positions $P = \{p_1, \ldots, p_m\}$ that contains positions of at least K distinct types. Replace these positions with ranks by putting $r_i := rank(p_i)$ and, thus, forming the set $R = \{r_1, \ldots, r_m\}$. Lemma 1 implies the equality

$$\big|lcp\big(\alpha[p_1..], \ldots, \alpha[p_m..]\big)\big| = \min_{R^- \le j < R^+} lcp(j)$$

where $R^- := \min R$ and $R^+ := \max R$.

Let us consider a segment $\Delta \subseteq [1, L]$ and call it *K-good* if for $i \in \Delta$ positions $SA(i)$ are of at least K distinct essential types. This enables us to restate Proposition 2 as follows:

Proposition 3. *The length of the longest common substring that appears in at least K input strings is equal to*

$$\max_{\Delta} \min_{\Delta^- \le j < \Delta^+} lcp(j)$$

where $\Delta = [\Delta^-, \Delta^+]$ *ranges over all K-good segments.*

This formula is already an improvement (compared to Proposition 2) since it only requires to consider a polynomial number of possibilities. Moreover, we shall indicate how maximum in Proposition 3 can be found in $O(L)$ time.

To this aim, note that if a K-good segment Δ is already considered then any $\Delta' \supset \Delta$ cannot give us a bigger value of minimum. For each i one may consider the segment $\Delta_i = [\Delta_i^-, \Delta_i^+]$ obeying the following properties:

- Δ_i starts at position i;
- Δ_i is K-good;
- Δ_i is the shortest segment obeying the above conditions.

Here index i ranges over $[1, L_0]$, where $L_0 \leq L$ is the smallest integer such that segment $[L_0 + 1, L]$ is not K-good.

Note that due to our assumption that sentinels are strictly less than normal letters, the first N elements of SA correspond to unessential positions occupied by the sentinels. The algorithm does not need to consider these positions and only examines segments $\Delta_{N+1}, \ldots, \Delta_{L_0}$.

Put

$$w(i) := \min_{\Delta_i^- \leq j < \Delta_i^+} lcp(j) \qquad \text{for all } N < i \leq L_0.$$

and consider the sequence:

$$w(N+1),\ w(N+2),\ \ldots,\ w(L_0) \tag{2}$$

Once the maximum among (2) is found, the problem is solved. We construct a pipeline with the first stage computing the sequence of segments

$$\Delta_{N+1},\ \Delta_{N+2},\ \ldots, \Delta_{L_0}$$

and the second stage calculating the respective minima (2)

Put $\Delta_i = [\Delta_i^-, \Delta_i^+]$. The first stage works as follows. It initially sets $\Delta_{N+1}^- := N + 1$ and then advances the right endpoint Δ_{N+1}^+ until getting a K-good segment. Then, on each subsequent iteration i it puts $\Delta_i^- := i$, $\Delta_i^+ := \Delta_{i-1}^+$ and again advances the right endpoint Δ_i^+ until Δ_i becomes K-good. (In case, no such segment can be extracted, it follows that $i > L_0$, so the end is reached.)

Lemma 3. *Δ_i is the shortest K-good segment starting at i.*

Proof. We claim that for any K-good segment $[i, j]$ one has $j \geq \Delta_{i-1}^+$. Indeed, suppose towards contradiction that $j < \Delta_{i-1}^+$. Since $[i, j]$ is K-good then so is $[i - 1, j]$. The latter, however, contradicts the minimality of Δ_{i-1}.

To test in $O(1)$ time if the current candidate forms a K-good segment the algorithm maintains an array of counters $c(1), \ldots, c(N)$. For each $N < j \leq L$ put $t(j)$ to be the type of position $SA(j)$ in α (recall that all these positions are essential). For each index i $(1 \leq i \leq N)$ the entry $c(i)$ stores the number of positions j in the current segment such that $t(j) = i$. Also, the number of non-zero entries of c (denoted by *npos*) is maintained.

Initializing c and $npos$ for Δ_{N+1} is trivial. Then, when the algorithm puts $\Delta_i^- = \Delta_{i-1}^- + 1$ it decrements the entry of c that corresponds to position $i-1$ (which has just been removed from the window) and adjusts $npos$, if necessary. Similarly, when the current segment is extended to the right, certain entries of c are increased and $npos$ is adjusted. To see whether the current segment is K-good one checks if $npos \geq K$. This completes the description of the first stage of the pipeline. Note that it totally takes $O(L)$ time.

The second stage aims to maintain the values (2) dynamically. This is done by the following (possibly folklore) trick. Consider a queue Q that, at any given moment i, holds the sequence of keys

$$lcp(\Delta_i^-),\ lcp(\Delta_i^- + 1),\ \ldots,\ lcp(\Delta_i^+ - 1)$$

Increasing index i the algorithm dequeues value $lcp(\Delta_i^-)$ from the head of Q (to account for the increase of Δ_i^-) and then enqueues some (possibly none) additional values to the tail of Q (to account for the increase of Δ_i^+, if any). The total number of these queue operations is $O(L)$.

We describe a method for maintaining the minimum of keys in Q under insertions and removals and serving each such request in $O(1)$ amortized time. A queue Q that holds a sequence $(q_1, \ldots, q_m)$ may be simulated by a pair of stacks S^1 and S^2. A generic configuration of these stacks during this simulation is as follows (here $0 \leq s \leq m$):

$$\begin{array}{l} S^1 = (\ q_s,\ \ q_{s-1}, \ldots, q_1\ \ ) \\ S^2 = (\ q_{s+1}, q_{s+2}, \ldots, q_m\ ) \end{array} \tag{3}$$

Here stack elements are listed from bottom to top. Initially Q is empty, hence so are S^1 and S^2. To enqueue a new key x (which becomes q_{m+1}) to Q one pushes x onto S^2. This takes $O(1)$ time. To dequeue q_1 from Q consider two cases. If $s > 0$ then S^1 is non-empty; pop the top element q_1 from S^1. Otherwise, one needs to transfer elements from S^2 to S^1. This is done by popping elements from S^2 one after another and simultaneously pushing them onto S^1 (in the same order). By (3) these operations preserve the order of keys in Q. Once they are complete, S^1 becomes non-empty and the first case applies. To estimate the running time note that any enqueued element may participate in an S^2-to-S^1 transfer at most once. Hence, an amortized bound of $O(1)$ follows.

Our algorithm simulates Q via S^1 and S^2, as explained above. Each S^i is additionally augmented to maintain the minimum among the keys it contains. This is achieved by keeping minima m^1, m^2 and a pair of auxiliary stacks M^1, M^2. When a new key x is pushed to S^i the algorithm saves the previous minimum m^i in M^i and updates m^i by $m^i := \min(m^i, x)$. When an element is popped from S^i the algorithm also pops m^i from M^i.

Altogether these manipulations with Q, S^i, M^i, and m^i take time that is proportional to the number of operations applied to Q. The latter is known to be $O(L)$. Hence, the algorithm computes the sequence of minima (2) and chooses the maximum (call it $M(K)$) among these values in $O(L)$ time, as claimed.

Let the above maximum be attained by a segment $\Delta = [\Delta^-, \Delta^+] \subseteq [N+1, L]$. Suppose that position $SA(\Delta^-)$ in string α corresponds to some position j in some input string α_i. Now the desired longest common substring is $\alpha_i[j..j+M(K)-1]$.

4 Acknowledgements

The authors are thankful to the students of Department of Mathematical Logic and Theory of Algorithms (Faculty of Mechanics and Mathematics, Moscow State University), and also to Maxim Ushakov and Victor Khimenko (Google Moscow) for discussions.

References

[AUH74] Alfred V. Aho, J. D. Ullman, and John E. Hopcroft. *The design and analysis of computer algorithms.* Addison-Wesley, Reading, MA, 1974.

[BFC00] M. Bender and M. Farach-Colton. The LCA problem revisited. In *LATIN '00: Proceedings of the 4th Latin American Symposium on Theoretical Informatics*, pages 88–94, 2000.

[CR03] Maxime Crochemore and Wojciech Rytter. *Jewels of stringology.* World Scientific Publishing Co. Inc., River Edge, NJ, 2003.

[Gus97] Dan Gusfield. *Algorithms on strings, trees, and sequences: computer science and computational biology.* Cambridge University Press, New York, NY, USA, 1997.

[KLA+01] Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikawa, and Kunsoo Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *CPM '01: Proceedings of the 12th Annual Symposium on Combinatorial Pattern Matching*, pages 181–192, London, UK, 2001. Springer-Verlag.

[KS03] J. Kärkkäinen and P. Sanders. Simple linear work suffix array construction. In *Proc. 13th International Conference on Automata, Languages and Programming.* Springer, 2003.

[MM90] Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. In *SODA '90: Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms*, pages 319–327, Philadelphia, PA, USA, 1990. Society for Industrial and Applied Mathematics.

[PST07] Simon J. Puglisi, W. F. Smyth, and Andrew H. Turpin. A taxonomy of suffix array construction algorithms. *ACM Comput. Surv.*, 39(2):4, 2007.

[Ukk95] Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.