Computing the longest common substring with one mismatch

Article *in* Problems of Information Transmission · March 2011

DOI: 10.1134/S0032946011010030

CITATIONS
11

READS
303

2 authors:

Maxim A. Babenko
National Research University Higher School of Economics
29 PUBLICATIONS **126** CITATIONS
SEE PROFILE

Tatiana Starikovskaya
Ecole Normale Supérieure de Paris
39 PUBLICATIONS **232** CITATIONS
SEE PROFILE

ISSN 0032-9460, Problems of Information Transmission, 2011, Vol. 47, No. 1, pp. 1–??. © Pleiades Publishing, Inc., 2011.
Original Russian Text © M.A. Babenko, T.A. Starikovskaya, 2011, published in Problemy Peredachi Informatsii, 2011, Vol. 47, No. 1, pp. 33–39.

LARGE SYSTEMS

Computing the Longest Common Substring with One Mismatch[1]

M. A. Babenko and T. A. Starikovskaya

Chair of Mathematical Logic and Theory of Algorithms, Faculty of Mechanics and Mathematics, Lomonosov Moscow State University

maxim.babenko@gmail.com tat.starikovskaya@gmail.com

Received May 7, 2010; in final form, August 24, 2010

Abstract—The paper describes an algorithm for computing longest common substrings of two strings α_1 and α_2 with one mismatch in $O(|\alpha_1||\alpha_2|)$ time and $O(|\alpha_1|)$ additional space. The algorithm always accesses symbols of α_2 *sequentially starting from the first symbol*. RAM-model of computation is used.

DOI: 10.1134/S003294601101????

1. INTRODUCTION

Longest common substring computing is one of the basic problems in stringology. However, for applications like bionformatics and text analysis the problem of computing longest common substrings with mismatches (insertions, deletions and substitutions of symbols) is of much bigger importance. There exist several approaches to this problem, e.g. dynamic programming. A solution based on dynamic programming computes longest common approximate substrings using time and space proportional to multiplication of the strings' lengths [**?**].

The problem solved in the paper can be formulated as follows.

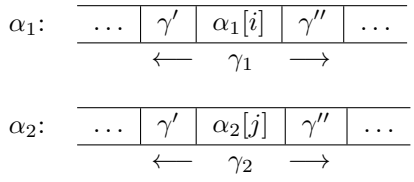
Given two strings α_1, α_2 compute all pairs of substrings of α_1 and α_2, which differ by at most one insertion, deletion or substitution of symbols and have maximal lengths (in case of insertions and deletions we define the biggest of the substrings' lengths to be the length of the longest common substring).

The developed algorithm uses the same time as the one based on dynamic programming and computes longest common substrings of two strings with one or zero mismatches. Suppose that the length of α_2 is sufficiently larger that the length of α_1. Additional memory used by the algorithm is then $O(|\alpha_1|)$. The algorithm reads α_2 for several times, but each time *sequentially from the left to the right, starting from the first symbol*. This condition is essential for applications which store α_2 in the external memory, because random access is costy in time in this case.

Preliminaries. We assume that a finite non-empty set Σ is fixed (an *alphabet*). Elements of this set are called *letters*. A finite ordered sequence of letters (possibly empty) is called a *string*. Letters of Σ 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 on Σ. We write $\alpha < \beta$ to denote that α lexicographically precedes β; similarly for other relation signs.

Letters in a string are numerated starting from 1; for example, symbols of a string α of length k are denoted by $\alpha[1], \ldots, \alpha[k]$. A *substring* of α from position i to position j (inclusively) is denoted by $\alpha[i..j]$.

[1] Supported in part by the Russian Foundation for Basic Research, project no. 09-01-00709-a.



Strings α_1, α_2 and their substrings γ_1, γ_2

Definition 1. A string $\beta = \alpha[1 : j]$ is a *prefix* of α and is denoted by $\alpha[: j]$. Similarly, $\beta = \alpha[i : |\alpha|]$ is a *suffix* of α and is denoted by $\alpha[i :]$.

The length of the longest common prefix of strings α and β and is denoted by $lcp(\alpha, \beta)$, and the length of the longest common suffix of strings α and β is denoted by $lcs(\alpha, \beta)$.

We assume the usual RAM model of computation [**?**].

First we give a naive algorithm for computing longest common substrings with one or zero substitutions of symbols (for deletions and insertions the idea is similar). Hereafter $|\alpha_1|$ is denoted by n_1 and $|\alpha_2|$ is denoted by n_2 ($n_2 \geq n_1$).

To compute the length L of the substrings the algorithm simply compares each substring of length d of the string α_1 with each substring of length d of the string α_2 symbol by symbol. If there exist a substring of α_1 and a substring of α_2, both of length d, differing in one or zero symbols, then the length of strings γ_1 and γ_2 is at least d. We run this procedure for all d from 1 to α_1 to obtain L.

After that we run the comparison for the second time and write out all substrings of length L differing in zero or one symbols.

Naive algorithm can be easily adjusted to read α_2 sequentially, starting from the first symbol. Namely, we fix a substring of length d of α_1 and compare it with strings $\alpha_2[1 : d]$, $\alpha_2[2 : d+1], \ldots,$ $\alpha_2[n_2 - d + 1 : n_2]$. Knowing symbols of $\alpha_2[i : i + d - 1]$ it is enough to read $\alpha_2[i + d]$ to get the substring $\alpha_2[i+1 : i+d]$. Therefore, we read the first d symbols of α_2 to get the substring $\alpha_2[1 : d]$ and then we read one symbol per step.

Running time of the naive algorithm is $O(n_1^2 n_2^2)$, the amount of additional memory used is $O(n_1)$. Our purpose is to speed up the algorithm in $n_1 n_2$ times.

2. THE ALGORITHM

We give a description of the algorithm computing the length of longest common substrings with one or zero substitutions. Similar to the naive algorithm, we can run it the second time to compute the substrings themselves. We explain how to change the algorithm to compute longest common substrings with insertions and deletions in Section **??**.

The idea of the developed algorithm is as follows. Suppose that γ_1 and γ_2 is one of the pairs of substrings we are looking for. Assume that γ_1 and γ_2 have equal parts γ', γ'' and there is only one position in which γ_1 and γ_2 might differ. Let this position coincide with position i in α_1 and with position j in α_2. Obviously, γ' is the longest common suffix of $\alpha_1[: i - 1]$ and $\alpha_2[: j - 1]$, and γ'' is the longest common prefix of $\alpha_1[i + 1 :]$ and $\alpha_2[j + 1 :]$ (fig. 1).

We denote $lcs(\alpha_1[: k], \alpha_2[: l])$ by $lcs(k, l)$ and $lcp(\alpha_1[k :], \alpha_2[l :])$ by $lcp(k, l)$. Assume that for positions i and j of strings α_1 and α_2, correspondingly, we know $lcs(i-1, j-1)$ and $lcp(i+1, j+1)$. Then $s(i, j) = lcs(i - 1, j - 1) + lcp(i + 1, j + 1) + 1$ is the length of the longest common substring of the strings α_1 and α_2 with a mismatch in positions i and j. We denote the length of the longest common substring with a mismatch in position i of string α_1 by d_i. Obviously, $d_i = \max\limits_{1 \leq j \leq n_2} s(i, j)$.

In fact, we do not check if symbols $\alpha_1[i]$ and $\alpha_2[j]$ are equal. We use position of a mismatch only as a label dividing γ_1 and γ_2 into two parts. Therefore, the algorithm computes the length of longest common substrings with zero or one substitutions of symbols.

Algorithm. Computing the length of longest common substring with one mismatch

MaxLength $\leftarrow 0$ $i \leftarrow 1 \dots n_1$ Build a suffix tree for $\alpha_1[1 : i-1]$ Build Z values for $\alpha_1[i+1 : n_1]$ Build Z values for $\alpha_1[1 : i-1]^{-1}$ $j \leftarrow 1 \dots n_2$ Compute $lcp(i+1, j+1)$ Compute $lcs(i-1, j-1)$ $s(i,j) \leftarrow 1 + lcp(i+1, j+1) + lcs(i-1, j-1)$ *MaxLength* $< s(i,j)$ *MaxLength* $\leftarrow s(i,j)$

We also give pseudocode of the algorithm. There *MaxLength* stands for the length of a longest common substring with one mismatch. The algorithm uses notions of suffix trees and Z values, which we define later.

The LCP algorithm computes $lcp(i+1, j+1)$ for a fixed i and all $j = 1, \dots, n_2$ and is described in Section 3. The LCS algorithm computes $lcs(i-1, j-1)$ for a fixed i and all $j = 1, \dots, n_2$ and is described in Section 4. Though they are described separately, in the main algorithm they are run simultaneously.

3. THE LCP ALGORITHM

In this section we give a general idea of the LCP algorithm. To compute $lcp(i+1, j)$ $j = 1, \dots, n_2$ the algorithm treats a string $\beta = \alpha_1[i+1 :]\$\alpha_2$, where $\$ \notin \Sigma$. We remind the definition of Z values of a string:

Definition 2. For a given string β and for any j, $1 < j \leq |\beta|$, $Z_j(\beta)$ is defined to be equal to $lcp(\beta, \beta[j :])$.

By the definition, $Z_{n_1+j-i}(\beta)$ is equal to $lcp(i+1, j)$. As it is shown in [?], Z values for β can be computed in $O(|\beta|) = O(n_2)$ time and $O(n_1 - i) = O(n_1)$ additional memory. Moreover, it is not necessary to store β in the memory explicitly.

4. THE LCS ALGORITHM

In this section we describe the LCS algorithm, which computes $lcs(i-1, j-1)$ for a fixed i and all $j = 1, \dots, n_2$. Note that it is impossible to use the LCP algorithm for reversed strings α_1 and α_2, as we want to read α_2 only from the left to the right, and, therefore, cannot reverse it.

Definition 3. *Pre-occurrence* of a string δ at position j in a string β is an occurrence of δ in β ending at position j.

Obviously, $lcs(i-1, j)$ is equal to the length of the longest suffix of the string $\alpha[: i-1]$ pre-occurring in α_2 at position j. Below we remind the definition of a suffix tree [?].

Definition 4. A *suffix tree* for a string β of length m is a directed tree with the following properties:

- A tree has m leaves, enumerated from 1 to m;
- Each non-leaf vertex has at least two children;
- Every arc is labeled with a non-empty substring of β;
- First symbols of labels of any two arcs outgoing from one vertex differ;
- On the path from the root to the leaf with number k a string $\beta[k :]\$$ is written.

Besides (explicit) vertexes mentioned in the definition, we also consider "*implicit*" vertexes, which are not stored explicitly but associated with their explicit parents and positions on arcs. An

arc with a label γ has $|\gamma| - 1$ implicit vertexes on it and we say that a vertex with number k has one outgoing arc labeled by $\gamma[k+1:]$.

For a compact representation arcs of a suffix tree are labeled not by substrings of β, but with corresponding endpoints of substrings. In this case, only linear memory is needed for storage of a suffix tree.

It is said that a vertex of a suffix tree is labeled by a string γ, if γ can be obtained by concatenating in order all labels on the path from the root to this vertex. Consider the following Lemma and Definition:

Lemma 1. *Existence of a vertex labeled by a string $x\gamma$($x \in \Sigma$, γ is a string) implies existence of a vertex labeled by γ, where both vertexes can either be explicit or implicit. Moreover, if the first vertex is explicit, then the second one is explicit as well.*

(for a proof see[?]).

Definition 5. A *suffix link* is a directed arc from an (explicit) vertex u labeled by a string $x\gamma$ ($x \in \Sigma$, γ is a string) to an (explicit) vertex labeled by γ.

E.i., Lemma **??** guarantees that each explicit vertex has a suffix link.

There are many suffix tree construction algorithms. One of the most famous is the algorithm by Ukkonen [**?**]. This algorithm is linear in time, and besides a suffix tree it also builds suffix links.

Now we return to the problem of longest common suffixes' computing. First we build a suffix tree T_i for $\alpha_1[:i-1]$ and compute Z values for $\alpha_1[:i-1]^{-1}$, where $\alpha_1[:i-1]^{-1} = \alpha_1[i-1]\alpha_1[i-2]\ldots\alpha_1[1]$.

Then we compute a pre-occurrence of a substring of $\alpha_1[:i-1]$ of maximal length at every position j of α_2, $1 \le j \le n_2$. For that we follow arcs of T_i starting from the root to read α_2. Suppose that we try to match $\alpha_2[k]$ and fail. This means that we are at a vertex v (either explicit or implicit) and there is no outgoing arc from this vertex with the label starting with $\alpha_2[k]$. Suppose that v is labeled by β. Then the next step is to *jump* to the vertex u labeled by $\beta[2:]$ (existence of u is guaranteed by Lemma **??**) as described below.

If v is explicit, the suffix link from v goes directly to u. If v is implicit, the procedure is a bit more complicated. First, we go up to the explicit parent of v. Then we take a suffix link from the parent and find ourselves somewhere on the path from the root to u. Note that there is only one path in T_i labeled by $\beta[2:]$ (see Definition **??**). So, to find u we just go down along the acrs checking the first symbols of arcs' labels. Namely, at each (explicit) vertex with the label ℓ we choose the arc with the label starting with $\beta[2+|\ell|:]$. The detailed description of this technique, called skip/count method, can be found in [?].

After arrival at u we proceed the traverse of T_i starting at $\alpha_2[k]$.

It follows from the traverse's description that the label $\alpha_1[p_j : t_j]$ of a vertex which we are at after matching position j of α_2 is *the longest substring* (not a suffix yet) of $\alpha_1[:i-1]$, pre-occurring at position j of α_2. Obviously, $lcs(\alpha_1[:i-1], \alpha_2[:j])$ is equal to $lcs(\alpha_1[:i-1], \alpha_1[p_j : t_j])$.

According to Definition **??**, the longest common suffix of strings $\alpha_1[:i-1]$ and $\alpha_1[:t_j]$ is equal to $k_j = Z_{i-t_j}(\alpha_1[1:i-1]^{-1})$. Then $\alpha_1[i-k_j : i-1]$ and, consequently, $\alpha_1[i-k_j+1 : i-1]$, $\alpha_1[i-k_j+2 : i-1]$,..., $\alpha_1[i-1]$ are suffixes of $\alpha_1[:t_j]$. Note that $\alpha_1[p_j : t_j]$ is also a suffix of $\alpha_1[:t_j]$. So, $lcs(\alpha_1[:i-1], \alpha_1[p_j : t_j]) = \min(k_j, t_j - p_j + 1)$, and, therefore, $lcs(\alpha_1[:i-1], \alpha_2[:j]) = \min(k_j, t_j - p_j + 1)$.

Time and space analysis. First we estimate time needed for the traverse of T_i. We estimate the number of jumps along suffix links, the number of operations made during skip/count steps and the number of letters' comparisons separately.

To prove that the number of jumps is less than $O(n_2)$, we use amortized analysis. Indeed, let j be the last symbol matched during the traverse and ℓ be length of the label of the current vertex. Then after a jump value $j - \ell$ will increase by one and it will not change if we follow α_2 down the arcs. As the maximum value of $j - \ell$ is n_2, the number of jumps is $O(n_2)$.

To estimate the number of operations made during skip/count steps, we again use amortized analysis. First we need one more definition and a lemma:

Definition 6. *Depth* of a vertex is the number of explicit vertexes on the path from the root of the tree to this vertex.

Lemma 2. *Let u, v be explicit vertexes of a suffix tree, and (u, v) be a suffix link from u to v. Then vertex depth of u exceeds vertex depth of v by at most one.*

(for a proof see[?]).

Current vertex depth can be decreased only when taking a suffix link. Lemma **??** guarantees that this step it can be decreased by at most two (by one when going up to the explicit parent and by one when taking a suffix link). Each operation during a skip/count step increases vertex depth by one. The maximal vertex depth is $O(n_1)$, therefore the total number of operations during skip/count steps is $O(2n_2 + n_1) = O(n_2)$.

So, the total number of operations made during the tree traverse (including letters' check) is $O(n_2)$.

Secondly, as have been already mentioned, Z values for $\alpha_1[:i-1]^{-1}$ and T_i can be computed in $O(n_1)$ time and space. So, the algorithm computes $lcs(i-1, j)$, $1 \le j \le n_2$ in time $O(n_2 + n_1) = O(n_2)$.

5. RESULTS

The main result of the paper is as follows.

Theorem 1. *The developed algorithm computes the longest common substring with one or zero substitutions of symbols in $O(n_1 n_2)$ time and $O(n_1)$ additional memory, $n_2 > n_1$. Moreover, it always accesses symbols of α_2 sequentially starting at the first symbol.*

Proof. For a fixed position i of α_1 it takes $O(n_2)$ time and $O(n_1)$ additional memory to compute the length of the longest common substring with a mismatch at position i. Indeed, Z values for strings $\alpha_1[:i-1]\$\alpha_2$ and $\alpha_1[:i-1]^{-1}$ can be computed in $O(n_2)$ time and $O(n_1)$ additional space. A suffix tree T_i with suffix links can be computed in $O(n_1)$ time and space. Finally, tree traverse takes $O(n_2)$ time to accomplish. So, for all n_1 positions of α_1 it takes $O(n_1 n_2)$ time and $O(n_1)$ additional memory.

Let us return to the case of one insertion or deletion of a symbol. To find the longest common substring with one insertion it is enough to compute $lcp(i+1, j+1) + lcs(i-1, j) + 1$ for all i, j and to find the maximum of these values. For deletion we compute the maximum of $lcp(i, j+1) + lcs(i, j-1) + 1$ for all i, j. Obviously, both maximums can be computed with the LCP and LCS algorithms. Moreover, same time and space bounds hold.

6. CONCLUSION

To conclude, the paper describes an algorithm for computing the longest common substring of two strings with one mismatch. The algorithm always accesses symbols of α_2 sequentially starting from the first symbol, which is important for applications. Though the algorithm hardly can be generalized for the case of k mismatches, the authors hope that the idea of the LCP algorithm can be used when the order of reading a string is essential.

The authors are thankful to Aleksei Lvovich Semenov for multiple and fruitful discussions.

REFERENCES

1. Gusfield, D., *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*, Cambridge: Cambridge Univ. Press, 1997. Translated under the title *Stroki, derev'ya i posledovatel'nosti v algoritmakh: informatika i vychislitel'naya biologiya*, St. Petersburg: Nevskii Dialekt, 2003.

2. Aho, A.V., Hopcroft, J.E., and Ullman, J.D., *The Design and Analysis of Computer Algorithms*, Reading: Addison-Wesley, 1976. Translated under the title *Postroenie i analiz vychislitel'nykh algoritmov*, Moscow: Williams, 2003.

3. Ukkonen, E., On-Line Construction of Suffix Trees, *Algorithmica*, 1995, vol. 14, no. 3, pp. 249–260.