

Longest Common Factor After One Edit Operation

Amihood Amir¹, Panagiotis Charalampopoulos², Costas S. Iliopoulos²,
Solon P. Pissis², and Jakub Radoszewski^{2,3(✉)}

¹ Department of Computer Science, Bar-Ilan University, Ramat Gan, Israel
amir@esc.biu.ac.il

² Department of Informatics, King's College London, London, UK
{panagiotis.charalampopoulos,costas.iliopoulos,solon.pissis}@kcl.ac.uk

³ Institute of Informatics, University of Warsaw, Warsaw, Poland
jrad@mimuw.edu.pl

Abstract. It is well known that the longest common factor (LCF) of two strings over an integer alphabet can be computed in time linear in the total length of the two strings. Our aim here is to present an algorithm that preprocesses two strings S and T in order to answer the following type of queries: Given a position i on S and a letter α , return an LCF of T and S' , where S' is the string resulting from S after substituting $S[i]$ with α . In what follows, we present an algorithm that, given two strings of length at most n , constructs in $\mathcal{O}(n \log^4 n)$ expected time a data structure of $\mathcal{O}(n \log^3 n)$ space that answers such queries in $\mathcal{O}(\log^3 n)$ time per query. After some trivial modifications, our approach can also support the case of single letter insertions or deletions in S .

Keywords: Longest common factor · Dynamic data structure · Suffix tree · Heavy-path decomposition · Orthogonal range searching

1 Introduction

In this work we consider strings over an integer alphabet. The longest common factor (LCF), also known as longest common substring, of two strings S and T , each of length at most n , can be computed in $\mathcal{O}(n)$ time [5, 11, 12, 15]. The LCF with k -mismatches problem has received much attention recently, in particular due to its applications in computational molecular biology [13, 17]. We refer the interested reader to [3, 8, 10, 14, 16].

A. Amir—Partially supported by the ISF grant 571/14 and the Royal Society.

C.S. Iliopoulos—Partially supported by the Onassis Foundation and the Royal Society.

J. Radoszewski—Supported by the “Algorithms for text processing with errors and uncertainties” project carried out within the HOMING programme of the Foundation for Polish Science co-financed by the European Union under the European Regional Development Fund.

Our motivation comes mainly from [14]; the author mentions that the solution to the LCF problem “is not robust and can vary significantly when the input strings are changed even by one letter”. Somewhat surprisingly, however, dynamic instances of the LCF problem have not yet been studied thoroughly to the best of our knowledge. In this paper, we aim at initiating a line of research on this general version of the problem, by presenting a solution for the restricted case, where any single edit operation is allowed. In what follows, we focus on the case of a letter substitution; insertions and deletions can be handled analogously.

Given two strings S and T over an integer alphabet, each of length at most n , one may ask the following question: How fast can we find an LCF of S and T after a single letter substitution? For instance, after substituting $S[i]$ with letter α . The goal is to preprocess S and T so that we do not need $\Omega(n)$ time to compute an LCF for each such query. A naïve solution is to precompute an LCF for all $\Theta(\sigma n)$ possible substitutions in $\Theta(\sigma n^2)$ time and then be able to answer any such query in $\mathcal{O}(1)$ time per query, where σ is the size of the alphabet.

Hence, for q such queries, computations can be done trivially in either $\mathcal{O}(qn)$ time (directly) or in $\mathcal{O}(\sigma n^2 + q)$ total time—this includes the $\mathcal{O}(\sigma n^2)$ time for preprocessing. We thus aim at an algorithm that will require $t_p = o(\sigma n^2)$ preprocessing time and $t_q = o(n)$ querying time. We will then be able to answer q such queries in $\mathcal{O}(t_p + qt_q)$ time, hence being more efficient than the aforementioned solutions, depending on the number q of queries to be answered.

Our Contribution. We present a data structure for solving the problem of LCF after a single letter substitution for two strings, each of length at most n , over an integer alphabet. Specifically, our construction requires $t_p = \mathcal{O}(n \log^4 n)$ expected preprocessing time and $\mathcal{O}(n \log^3 n)$ space. After this preprocessing, the answer to any subsequent query for i and α is computed in $t_q = \mathcal{O}(\log^3 n)$ time.

2 Preliminaries

We begin with basic definitions and notation generally following [6]. Let $S = S[1]S[2] \dots S[n]$ be a *string* of length $|S| = n$ over a finite ordered alphabet. We consider *integer alphabets*, i.e. Σ of size $|\Sigma| = \sigma = n^{\mathcal{O}(1)}$. By ε we denote an *empty string*. For two positions i and j on S , we denote by $S[i \dots j] = S[i] \dots S[j]$ the *factor* (sometimes called *substring*) of S that starts at position i and ends at position j (it equals ε if $j < i$). We recall that a *prefix* of S is a factor that starts at position 1 ($S[1 \dots j]$) and a *suffix* is a factor that ends at position n ($S[i \dots n]$). We denote the *reverse string* of S by S^R , i.e. $S^R = S[n]S[n-1] \dots S[1]$.

Let Y be a string of length m with $0 < m \leq n$. We say that there exists an *occurrence* of Y in S , or, more simply, that Y *occurs in* S , when Y is a factor of S . Every occurrence of Y can be characterised by a starting position in S . We thus say that Y occurs at the *starting position* i in S when $Y = S[i \dots i+m-1]$.

Given two strings S and T , a string Y that occurs in both is a *longest common factor* (LCF) of S and T if there is no longer factor of T that is also a factor of S ; note that S and T can have multiple LCFs. We introduce a natural representation of an LCF of S and T as a triple (m, p, q) such that

$S[p..p+m-1] = T[q..q+m-1]$ is an LCF of S and T . The problem in scope can be formally defined as follows; see also Table 1 for an example.

LCF AFTER ONE SUBSTITUTION

Input: Two strings S and T .

Query: $\text{LCF}(i, \alpha)$ that represents an LCF of S' and T , where $S'[i] = \alpha$ and $S'[j] = S[j]$, for all $1 \leq j \leq |S|$, $j \neq i$.

Table 1. Answers to all $\text{LCF}(i, \alpha)$ queries for $S = \text{baccb}$ and $T = \text{baacca}$ over alphabet $\Sigma = \{\mathbf{a}, \mathbf{b}, \mathbf{c}\}$. In each case the corresponding LCF string is shown.

α	i					
	1	2	3	4	5	
a	(4,1,2) aacc	(3,2,3) acc	(4,1,1) baac	(2,3,5) ca	(4,2,3) acca	
b	(3,2,3) acc	(2,3,4) cc	(2,1,1) ba	(2,2,3) ac	(3,2,3) acc	
c	(3,2,3) acc	(2,3,4) cc	(3,2,3) acc	(3,2,3) acc	(3,2,3) acc	

Suffix Tree and Suffix Array. The *suffix tree* $\mathcal{T}(S)$ of a non-empty string S of length n is a compact trie representing all suffixes of S . The *branching* nodes of the trie as well as the *terminal* nodes, that correspond to suffixes of S , become *explicit* nodes of the suffix tree, while the other nodes are *implicit*. Each edge of the suffix tree can be viewed as an upward maximal path of implicit nodes starting with an explicit node. Moreover, each node belongs to a unique path of that kind. Thus, each node of the trie can be represented in the suffix tree by the edge it belongs to and an index within the corresponding path. We let $\mathcal{L}(v)$ denote the *path-label* of a node v , i.e., the concatenation of the edge labels along the path from the root to v . We say that v is path-labelled $\mathcal{L}(v)$. Additionally, $\mathcal{D}(v) = |\mathcal{L}(v)|$ is used to denote the *string-depth* of node v . A terminal node v such that $\mathcal{L}(v) = S[i..n]$ for some $1 \leq i \leq n$ is also labelled with index i . It should be clear that each factor of S is uniquely represented by either an explicit or an implicit node of $\mathcal{T}(S)$, called its *locus*. In standard suffix tree implementations, we assume that each node of the suffix tree is able to access its parent. Once $\mathcal{T}(S)$ is constructed, it can be traversed in a depth-first manner to compute the string-depth $\mathcal{D}(v)$ for each node v . It is known that the suffix tree of a string of length n , over an integer ordered alphabet, can be computed in time and space $\mathcal{O}(n)$ [7]. In the case of integer alphabets, in order to access the child of an explicit node by the first letter of its edge label in $\mathcal{O}(1)$ time, perfect hashing [9] can be used.

The *suffix array* of a non-empty string S of length n , denoted by $\text{SA}(S)$, is an integer array of size $n+1$ storing the starting positions of all (lexicographically) sorted suffixes of S , i.e. for all $1 < r \leq n+1$ we have $S[\text{SA}(S)[r-1]..n] < S[\text{SA}(S)[r]..n]$. Note that we explicitly add the empty suffix to the array. The suffix array $\text{SA}(S)$ corresponds to a pre-order traversal of all terminal nodes

of the suffix tree $\mathcal{T}(S)$. The inverse $\text{iSA}(S)$ of the array $\text{SA}(S)$ is defined by $\text{iSA}(S)[\text{SA}(S)[r]] = r$, for all $1 \leq r \leq n + 1$.

Algorithmic Tools for Trees. Let \mathcal{T} be a rooted tree with integer weights on nodes. We require that the weight of the root is zero and the weight of any other node is strictly greater than the weight of its parent. We say that a node v is a *weighted ancestor* of a node u at depth ℓ if v is the highest ancestor of u with weight of at least ℓ .

Lemma 1 ([2]). *After $\mathcal{O}(n)$ -time preprocessing, weighted ancestor queries for nodes of a tree \mathcal{T} of size n can be answered in $\mathcal{O}(\log \log n)$ time per query.*

The following corollary applies Lemma 1 to the suffix tree.

Corollary 2. *The locus of any factor $S[i..j]$ in $\mathcal{T}(S)$ can be computed in $\mathcal{O}(\log \log n)$ time after $\mathcal{O}(n)$ -time preprocessing.*

Let us also recall the notion of heavy-path decomposition. Consider a rooted tree \mathcal{T} . For each non-leaf node u of \mathcal{T} , the *heavy edge* (u, v) is an edge for which the subtree rooted at v has the maximal number of leaves (in case of several such subtrees, we fix one of them). A *heavy path* is a maximal path of heavy edges.

Let π be a heavy path and u be its topmost node. Assume that u contains m leaves in its subtree. We then denote by $L(\pi)$ the level of the path π , which is equal to $\log m$ ¹. The crucial property of heavy-path decompositions can be stated as follows.

Observation 1. *For any leaf v of \mathcal{T} , the levels of all heavy paths visited on the path from v to the root of \mathcal{T} are distinct.*

Range Maxima in 2-d. Assume we are given a collection \mathcal{P} of n points in a 2-d grid with integer weights of magnitude $\mathcal{O}(n)$. In a 2-d range maximum query $\text{RMQ}(\mathcal{P}, [a, b] \times [c, d])$, given a rectangle $[a, b] \times [c, d]$, we are to report the maximum weight of a point from \mathcal{P} in the rectangle. We assume that the point that attains this maximum is also computed.

Lemma 3 ([1]). *Range maximum queries over a set of n weighted points in 2-d can be answered in $\mathcal{O}(\log n)$ time with a data structure of size $\mathcal{O}(n \log n)$ that can be constructed in $\mathcal{O}(n \log^2 n)$ expected time.*

Among orthogonal range searching problems one can also consider the so-called *range emptiness queries*, in which we are only to check if any of the n points is located inside a query rectangle. Such queries are obviously a special case of 2-d range maximum queries.

¹ Throughout the paper we assume that $\log m$ denotes the binary logarithm of m rounded down to the nearest integer.

3 Two Auxiliary Problems

We assume throughout the paper that both strings S and T are over an integer alphabet Σ and that each of them has length at most n . We can decompose the problem in scope into the following two subproblems; we then only need to take the maximum. See also Tables 2 and 3 for an example.

LCF AVOIDING i

Input: Two strings S and T .

Output: An array LCF_1 of size $|S|$ such that $\text{LCF}_1[i]$ represents a longest factor Y common to S and T such that Y occurs in S at some position p , where $p \leq i - |Y|$ or $p > i$.

Table 2. The $\text{LCF}_1[i]$ array for $S = \text{baccb}$ and $T = \text{baacca}$. The auxiliary arrays that are used to compute it in Sect. 4 are also presented.

i	1	2	3	4	5
$\text{LCF}_1[i]$	(3,2,3) acc	(2,3,4) cc	(2,1,1) ba	(2,2,3) ac	(3,2,3) acc
$\overleftarrow{\text{LCF}}_1[i]$	(1,1,1) b	(2,1,1) ba	(2,2,3) ac	(3,2,3) acc	
$\text{LCF}_1^2[i]$		(3,2,3) acc	(2,3,4) cc	(1,5,1) b	(1,5,1) b

LCF INCLUDING $S[i] := \alpha$

Input: Two strings S and T .

Query: $\text{LCF}_2(i, \alpha)$ that represents an LCF Y of S' and T , where $S'[i] = \alpha$ and $S'[j] = S[j]$, for all $1 \leq j \leq |S|$, $j \neq i$, such that Y occurs in S' at some position $p \in \{i - |Y| + 1, \dots, i\}$.

Table 3. Answers to all $\text{LCF}_2(i, \alpha)$ queries for $S = \text{baccb}$ and $T = \text{baacca}$.

α	i				
	1	2	3	4	5
a	(4,1,2) aacc	(3,2,3) acc	(4,1,1) baac	(2,3,5) ca	(4,2,3) acca
b	(2,1,1) ba	(1,2,1) b	(1,3,1) b	(1,4,1) b	(1,5,1) b
c	(2,1,5) ca	(2,2,4) cc	(3,2,3) acc	(3,2,3) acc	(2,4,4) cc

Observation 2. Suppose that we replace $S[i]$ by α . If this gives us a longer common factor than an LCF of S and T , this has to contain position i of S .

We first build $\mathcal{T}(X)$, where $X = T\#S$ and $\# \notin \Sigma$ is a letter that is lexicographically smaller than all the letters of Σ . We then store for every node of $\mathcal{T}(X)$ whether it has descendants from S , T , or both and a starting position

in each case; we can do this by performing a depth-first traversal. We further preprocess $\mathcal{T}(X)$ in $\mathcal{O}(n)$ time so that we can answer *lowest common ancestor* (LCA) queries for any pair of explicit nodes in $\mathcal{O}(1)$ time per query [4]. We construct and preprocess in the same manner the suffix tree $\mathcal{T}(X^R)$, where $X^R = S^R \# T^R$. There will be more preprocessing that will be described later.

4 LCF Avoiding i

In this section we present an algorithm for determining an LCF of S and T avoiding position i in S . It is clear that this is the longest between an LCF of $S[1..i-1]$ and T and an LCF of $S[i+1..|S|]$ and T . Let us denote the representation of the former by $\overleftarrow{\text{LCF}}_1[i-1]$ and the representation of the latter by $\overrightarrow{\text{LCF}}_1[i+1]$; see also Table 2 for an example. We will show how to efficiently compute $\overrightarrow{\text{LCF}}_1[i]$ for all $i = |S|, \dots, 2$.

We denote the length of the *longest common prefix* of two strings W and Y by $\text{lcp}(W, Y)$. Further we will also use the notation $\text{lcs}(W, Y)$ to denote the length of the longest common suffix of W and Y . Let us make the following observation.

Observation 3. *If for a pair (p, q) with $i \leq p \leq |S|$ and $1 \leq q \leq |T|$ we have*

$$m = \text{lcp}(S[p..|S|], T[q..|T|]) = \max_{\substack{i \leq j \leq |S| \\ 1 \leq k \leq |T|}} \{\text{lcp}(S[j..|S|], T[k..|T|])\},$$

then $S[p..p+m-1] = T[q..q+m-1]$ is an LCF of $S[i..|S|]$ and T .

We first traverse $\mathcal{T}(X)$ in a depth-first manner in order to store, for every explicit node u , the maximal length $\ell(u)$ of the longest common prefix of $\mathcal{L}(u)$ and any suffix of T and a position $t(u)$ of T where the maximum is attained. If a node u has descendants from T , then clearly $\ell(u) = \mathcal{D}(u)$ and $t(u)$ is already stored. Whenever we reach a node u that does not have descendants from T , we set the values $\ell(u)$ and $t(u)$ equal to these of u 's explicit parent.

To compute the array $\overrightarrow{\text{LCF}}_1$, we go through the terminal nodes of $\mathcal{T}(X)$ that represent suffixes of S in increasing order with regards to the length of the suffix they represent. We initialize variables $\text{lcf} = p = q = 0$. When processing node u , with $\mathcal{L}(u) = S[i..|S|]$, we first check whether $\ell(u) > \text{lcf}$; if so, we set $\text{lcf} = \ell(u)$, $p = i$ and $q = t(u)$. Then, based on Observation 3, we set $\overrightarrow{\text{LCF}}_1[i] = (\text{lcf}, p, q)$.

The computation of $\overleftarrow{\text{LCF}}_1[i]$ for $i = 1, \dots, |S| - 1$ can be done in a symmetric way by employing $\mathcal{T}(X^R)$. Finally, we compare $\overleftarrow{\text{LCF}}_1[i-1]$ with $\overrightarrow{\text{LCF}}_1[i+1]$ and store the longer one as $\text{LCF}_1[i]$. We thus arrive at the following result.

Lemma 4. *Problem LCF AVOIDING i can be solved in $\mathcal{O}(n)$ time and space.*

5 LCF Including $S[i] := \alpha$

We first compute two factors, P and Q , of T :

- P is the longest factor of T that is equal to a suffix of $S[1 \dots i-1]\alpha$;
- Q is the longest factor of T that is equal to a prefix of $S[i+1 \dots |S|]$.

In addition to P and Q , we will also compute the locus p of P^R in $\mathcal{T}(X^R)$ and the locus q of Q in $\mathcal{T}(X)$. Note that q is an explicit node of $\mathcal{T}(X)$, however, p need not be an explicit node of $\mathcal{T}(X^R)$. We first compute the locus p' of P in $\mathcal{T}(X)$ (which may be implicit as well).

The locus p' is computed by performing binary search on $S[1 \dots i-1]$. We first identify the locus of $S[\lfloor i/2 \rfloor \dots i-1]$ in $\mathcal{T}(X)$. If it is explicit, we check whether it has an outgoing edge with label α and if the explicit node we obtain by following this edge has descendants in T . If it is implicit, we check if the next letter on the path-label of the edge is α and whether the explicit node to which this edge points has descendants in T . If the corresponding check succeeds, we look at the locus of $S[\lfloor i/4 \rfloor \dots i-1]$, otherwise we look at the locus of $S[\lfloor 3i/4 \rfloor \dots i-1]$; and so on. The whole binary search works in $\mathcal{O}(\log n \log \log n)$ time using Corollary 2.

Recall that for the closest explicit descendant of p' we store the starting position of some occurrence of the corresponding factor in X . We can then use this information to find the locus p of P^R in $\mathcal{T}(X^R)$ in $\mathcal{O}(\log \log n)$ time using Corollary 2.

Finally, the locus q of Q in $\mathcal{T}(X)$ can be analogously computed by binary search on $S[i+1 \dots |S|]$ in $\mathcal{O}(\log n \log \log n)$ time.

Let us note that $\text{LCF}_2(i, \alpha)$ corresponds to the longest factor of T that is composed by concatenating a suffix of P with a prefix of Q . We say that a node u of $\mathcal{T}(X^R)$ with path-label U and a node v of $\mathcal{T}(X)$ with path-label V are T -attached if and only if $U^R V$ is a factor of T . We thus aim at finding an ancestor u of p in $\mathcal{T}(X^R)$ and an ancestor v of q in $\mathcal{T}(X)$ such that u and v are T -attached and the sum $\mathcal{D}(u) + \mathcal{D}(v)$ of string-depths is maximal.

5.1 $\tilde{\mathcal{O}}(|P|)$ -Time Query

In this section we show how to find the desired pair of T -attached nodes (u, v) in $\tilde{\mathcal{O}}(\mathcal{D}(p)) = \tilde{\mathcal{O}}(|P|)$ time². We improve this solution in the next subsection.

Recall that $\text{SA}(T)$ and $\text{SA}(T^R)$ are the suffix arrays of T and T^R , respectively. Note that $\text{SA}(T)$ (resp. $\text{SA}(T^R)$) corresponds to a pre-order traversal of all the terminal nodes of $\mathcal{T}(X)$ ($\mathcal{T}(X^R)$) that are loci of suffixes of T concatenated with $\#S$ (loci of suffixes of T^R). For each explicit node v of $\mathcal{T}(X)$ we precompute a range $\text{range}_T(v)$ of $\text{SA}(T)$ that corresponds to suffixes of T that start with $\mathcal{L}(v)$. Similarly, for each explicit node u of $\mathcal{T}(X^R)$ we precompute a range $\text{range}_{T^R}(u)$ of $\text{SA}(T^R)$ that corresponds to suffixes of T^R that start with $\mathcal{L}(u)$. The precomputations can be done via bottom-up traversals of $\mathcal{T}(X)$ and $\mathcal{T}(X^R)$ in $\mathcal{O}(n)$ time. For $\mathcal{T}(X)$, the range of every explicit node is computed by summing the

² Throughout the paper $\tilde{\mathcal{O}}$ notation suppresses $\log^{\mathcal{O}(1)} n$ factors.

ranges of its explicit children. Additionally, for a terminal node being a locus of a suffix $T[j..|T|]\#S$ we extend its range by the element $\text{iSA}(T)[j]$. The computations for $\mathcal{T}(X^R)$ are analogous. The ranges of implicit nodes of $\mathcal{T}(X)$ and $\mathcal{T}(X^R)$ are defined as the corresponding ranges of their closest explicit descendants.

We can use the ranges to state an equivalent condition on when two nodes are T -attached:

Observation 4. *Node u of $\mathcal{T}(X^R)$ and node v of $\mathcal{T}(X)$ are T -attached if and only if there exist integers $i \in \text{range}_{T^R}(u)$ and $j \in \text{range}_T(v)$ such that $\text{SA}(T^R)[i] = |T| + 2 - \text{SA}(T)[j]$.*

It turns out that the problem of checking if two nodes are T -attached can be reduced to a 2-d range emptiness query. Indeed, let us consider a $(|T| + 1) \times (|T| + 1)$ grid. We create a collection \mathcal{P} of $|T| + 1$ points from the grid; for each position j in T , $j \in \{1, \dots, |T| + 1\}$, we select the point:

$$(\text{iSA}(T^R)[|T| + 2 - j], \text{iSA}(T)[j]).$$

Intuitively, the dimensions of the grid correspond to $\text{SA}(T^R)$ and $\text{SA}(T)$ and the points that are selected correspond to pairs of suffixes: $T^R[|T| + 2 - j..|T|] = (T[1..j-1])^R$ and $T[j..|T|]$. Observation 5 is a reformulation of Observation 4 in terms of range emptiness queries in \mathcal{P} .

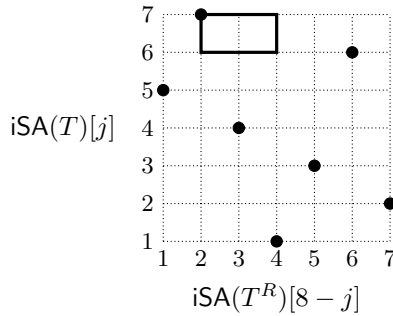
Observation 5. *Node u of $\mathcal{T}(X^R)$ and node v of $\mathcal{T}(X)$ are T -attached if and only if the rectangle $\text{range}_{T^R}(u) \times \text{range}_T(v)$ contains a point from \mathcal{P} .*

Example 5. Consider the string $T = \text{baacca}$ from Tables 1, 2 and 3. Then:

$$\text{SA}(T^R) = 7 (\varepsilon), 4 (\text{aab}), 5 (\text{ab}), 1 (\text{accaab}), 6 (\text{b}), 3 (\text{caab}), 2 (\text{ccaab})$$

$$\text{SA}(T) = 7 (\varepsilon), 6 (\text{a}), 2 (\text{aacca}), 3 (\text{acca}), 1 (\text{baacca}), 5 (\text{ca}), 4 (\text{cca})$$

The points from the set \mathcal{P} on the 7×7 grid are shown on the figure below.



Consider node u of $\mathcal{T}(X^R)$ such that $\mathcal{L}(u) = \text{a}$ and node v of $\mathcal{T}(X)$ such that $\mathcal{L}(v) = \text{c}$. Then $\text{range}_{T^R}(u) = [2, 4]$ and $\text{range}_T(v) = [6, 7]$. The corresponding rectangle $\text{range}_{T^R}(u) \times \text{range}_T(v)$ contains a single point $(2, 7)$ from \mathcal{P} which corresponds to the second suffix in $\text{SA}(T^R)$, which is $Y = \text{aab}$, and the seventh suffix in $\text{SA}(T)$, which is $Z = \text{cca}$. Note that, indeed, Y starts with an a , Z starts with a c and $Y^R Z = T$. Hence, u and v are T -attached.

From the previous subsection we know that we need to find an ancestor u of p in $\mathcal{T}(X^R)$ and an ancestor v of q in $\mathcal{T}(X)$ such that u and v are T -attached and $\mathcal{D}(u) + \mathcal{D}(v)$ is maximal. To find the desired nodes, we examine each node u on the path from p to the root of $\mathcal{T}(X^R)$ and apply binary search to find the deepest node v on the path from q to the root of $\mathcal{T}(X)$ such that u and v are T -attached. There are $|P|$ binary searches to perform, each of which has $\mathcal{O}(\log n)$ steps. In each such step we first locate the required node in $\mathcal{O}(\log \log n)$ time by Corollary 2 and then check if the two nodes are T -attached via Observation 5 using a range emptiness query which takes $\mathcal{O}(\log n)$ time by Lemma 3. In total, we arrive at an $\mathcal{O}(|P| \log^2 n)$ -time computation of $\text{LCF}_2(i, \alpha)$.

5.2 $\tilde{\mathcal{O}}(1)$ -Time Query

Main Idea. In order to drop the $|P|$ factor from the complexity, we make use of the heavy-path decompositions of $\mathcal{T}(X^R)$ and $\mathcal{T}(X)$. For each heavy path, we store its level. Moreover, each explicit node w of $\mathcal{T}(X^R)$ and $\mathcal{T}(X)$ stores the topmost node $\text{top}(w)$ of its heavy path. For simplicity we first assume that p is explicit in $\mathcal{T}(X^R)$ (recall that q is explicit in $\mathcal{T}(X)$); we then discuss how to tackle the case of p being implicit. By Observation 4, the sought node v is always explicit and u may be implicit only if p is implicit and u is on the same edge of $\mathcal{T}(X^R)$ as p .

The path from p to the root of $\mathcal{T}(X^R)$ is composed of *prefix fragments* of at most $\log n + 1$ heavy paths interleaved by single non-heavy (compact) edges. Here a prefix fragment of a path π is a path connecting the topmost node of π with any of its explicit nodes. We denote this decomposition by $H(p)$; note that it can be computed in $\mathcal{O}(\log n)$ time by using the *top*-pointers of nodes, starting from p . Similarly, we can decompose the path from q to the root of $\mathcal{T}(X)$ into a collection $H(q)$ of at most $\log n + 1$ prefix fragments of heavy paths in $\mathcal{O}(\log n)$ time. For each of the $\mathcal{O}(\log^2 n)$ pairs of prefix fragments of heavy paths $\pi'_1 \in H(p)$ and $\pi'_2 \in H(q)$ we will check if there are any T -attached nodes $u \in \pi'_1$ and $v \in \pi'_2$ and, if so, find the maximum value of $\mathcal{D}(u) + \mathcal{D}(v)$ among such pairs.

Precomputations. We consider the same 2-d grid as described in the previous subsection with $\mathcal{O}(\log^2 n)$ collections of points being copies of the collection \mathcal{P} ; they are denoted by $\mathcal{P}_{a,b}^{(I)}$, $\mathcal{P}_a^{(II)}$, $\mathcal{P}_b^{(III)}$, $\mathcal{P}^{(IV)}$ for $a, b = 0, \dots, \log n$. The points in the respective collections have the following weights:

- $(j, k) \in \mathcal{P}_{a,b}^{(I)}$: $\mathcal{D}(u) + \mathcal{D}(v)$ where u is the lowest node on a heavy path of level a in $\mathcal{T}(X^R)$ such that $j \in \text{range}_{T^R}(u)$ and v is the lowest node on a heavy path of level b in $\mathcal{T}(X)$ such that $k \in \text{range}_T(v)$;
- $(j, k) \in \mathcal{P}_a^{(II)}$: $\mathcal{D}(u)$ where u is the lowest node on a heavy path of level a in $\mathcal{T}(X^R)$ such that $j \in \text{range}_{T^R}(u)$;
- $(j, k) \in \mathcal{P}_b^{(III)}$: $\mathcal{D}(v)$ where v is the lowest node on a heavy path of level b in $\mathcal{T}(X)$ such that $k \in \text{range}_T(v)$;
- each point in $\mathcal{P}^{(IV)}$ has a unit weight.

By Observation 1, the heavy paths in each case, if they exist, are determined in a unique way by j and k . If any of the nodes u or v does not exist, we set its depth \mathcal{D} to $-\infty$. Note that each of the nodes u and v , if it exists, is explicit in $\mathcal{T}(X^R)$ and $\mathcal{T}(X)$, respectively.

The total size of the collections of points is $\mathcal{O}(n \log^2 n)$. We further have:

Lemma 6. *Weights of the points from the collections $\mathcal{P}_{a,b}^{(I)}$, $\mathcal{P}_a^{(II)}$, $\mathcal{P}_b^{(III)}$, $\mathcal{P}^{(IV)}$ for $a, b = 0, \dots, \log n$ can be computed in $\mathcal{O}(n \log^2 n)$ time.*

Proof. First, for each $b = 0, \dots, \log n$ and $k = 1, \dots, |T|+1$ we compute $D[b][k] = \mathcal{D}(v)$ where v is the lowest node on a heavy path of level b in $\mathcal{T}(X)$ such that $k \in \text{range}_T(v)$. For each explicit node w of $\mathcal{T}(X)$ we consider its heavy edge (if exists) that leads to its explicit child w' and for each $k \in \text{range}_T(w) \setminus \text{range}_T(w')$, we set $D[b][k] = \mathcal{D}(w)$ where b is the level of the heavy path that contains the node w . This computation works in $\mathcal{O}(n \log n)$ time.

In the same way we can compute a symmetric array $D'[a][j] = \mathcal{D}(u)$ where u is the lowest node on a heavy path of level a in $\mathcal{T}(X^R)$ such that $j \in \text{range}_{T^R}(u)$. The two arrays allow us to compute the weights of all points. For example, the weight of the point $(j, k) \in \mathcal{P}_{a,b}^{(I)}$ is $D'[a][j] + D[b][k]$. \square

Queries. Let us consider a heavy path π_1 of level a in $\mathcal{T}(X^R)$ and a heavy path π_2 of level b in $\mathcal{T}(X)$. Let π'_1 be a prefix fragment of π_1 that leads from node x_1 down to node y_1 and π'_2 be a prefix fragment of π_2 that leads from node x_2 down to node y_2 . Let A_1, B_1, C_1 be intervals such that

$$\text{range}_{T^R}(y_1) = B_1 \quad \text{and} \quad \text{range}_{T^R}(x_1) \setminus B_1 = A_1 \cup C_1.$$

Similarly, we define the intervals A_2, B_2, C_2 so that:

$$\text{range}_T(y_2) = B_2 \quad \text{and} \quad \text{range}_T(x_2) \setminus B_2 = A_2 \cup C_2;$$

see Fig. 1 for an illustration. Then the maximum of $\mathcal{D}(u) + \mathcal{D}(v)$ over all T -attached pairs of nodes $u \in \pi'_1$ and $v \in \pi'_2$ is the maximum of the following nine values:

- (1) $\text{RMQ}(\mathcal{P}_{a,b}^{(I)}, A_1 \times A_2)$
- (2) $\text{RMQ}(\mathcal{P}_{a,b}^{(I)}, A_1 \times C_2)$
- (3) $\text{RMQ}(\mathcal{P}_{a,b}^{(I)}, C_1 \times A_2)$
- (4) $\text{RMQ}(\mathcal{P}_{a,b}^{(I)}, C_1 \times C_2)$
- (5) $\text{RMQ}(\mathcal{P}_a^{(II)}, A_1 \times B_2) + \mathcal{D}(y_2)$
- (6) $\text{RMQ}(\mathcal{P}_a^{(II)}, C_1 \times B_2) + \mathcal{D}(y_2)$
- (7) $\mathcal{D}(y_1) + \text{RMQ}(\mathcal{P}_b^{(III)}, B_1 \times A_2)$
- (8) $\mathcal{D}(y_1) + \text{RMQ}(\mathcal{P}_b^{(III)}, B_1 \times C_2)$
- (9) $\mathcal{D}(y_1) + \mathcal{D}(y_2)$ if $\text{RMQ}(\mathcal{P}^{(IV)}, B_1 \times B_2) \neq -\infty$.

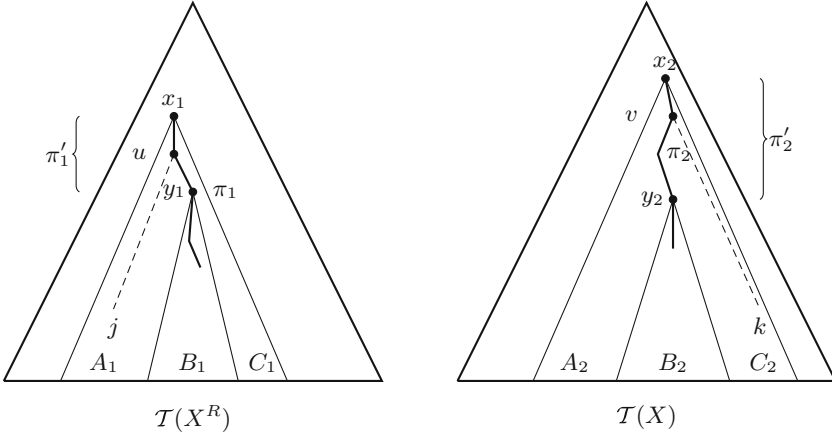


Fig. 1. Illustration of the notations used to handle a pair of prefix fragments $\pi'_1 \in H(p)$ and $\pi'_2 \in H(q)$. Assume that the sought pair of T -attached nodes $u \in \pi'_1$ and $v \in \pi'_2$ are located as shown. Here $j \in \{1, \dots, |T| + 1\}$ is an index for which u is the lowest node on π_1 such that $j \in \text{range}_{T^R}(u)$; same holds for k and $v \in \pi_2$. Then $\mathcal{D}(u) + \mathcal{D}(v)$ is computed by value (2): $\text{RMQ}(\mathcal{P}_{a,b}^{(I)}, A_1 \times C_2)$.

Values (1)–(4) correspond to the situation when the sought nodes u and v are located strictly above y_1 and y_2 , respectively. Values (5)–(6) assume the case that $v = y_2$; values (7)–(8) assume the case that $u = y_1$; finally, value (9) assumes that $u = y_1$ and $v = y_2$.

The maximum of the values of the form (1)–(9) is computed for all pairs of prefix fragments of heavy paths $\pi'_1 \in H(p)$ and $\pi'_2 \in H(q)$. The global maximum is the length of an LCF with $S[i] := \alpha$. Its example occurrence can be retrieved from the coordinates (j, k) of the point for which the range maximum is obtained. Indeed, let $r = \text{SA}(T)[k]$. Then an LCF occurs in S' and T at positions $i - d$ and $r - 1 - d$, respectively, where $d = \text{lcs}(S[1 \dots i - 1], T[1 \dots r - 2])$. Note that d can be computed via an LCA query in $\mathcal{T}(X^R)$ in $\mathcal{O}(1)$ time.

The Case of Implicit p . If p is not explicit, we make the above computations for the nearest explicit ancestor of p . We need to consider separately the case that u is an implicit node located between p and this ancestor. In this case $\text{range}_{T^R}(u) = \text{range}_{T^R}(p)$; we denote this interval by F . Hence, we take $u = p$. We consider every prefix fragment $\pi'_2 \in H(q)$ of a heavy path with level b , endpoints x_2, y_2 , and implied intervals A_2, B_2, C_2 and pick the maximum of:

- $\mathcal{D}(p) + \text{RMQ}(\mathcal{P}_b^{(III)}, F \times A_2)$
- $\mathcal{D}(p) + \text{RMQ}(\mathcal{P}_b^{(III)}, F \times C_2)$
- $\mathcal{D}(p) + \mathcal{D}(y_2)$ if $\text{RMQ}(\mathcal{P}^{(IV)}, F \times B_2) \neq -\infty$.

Lemma 7. After $\mathcal{O}(n \log^4 n)$ expected time and $\mathcal{O}(n \log^3 n)$ space preprocessing, LCF INCLUDING $S[i] := \alpha$ queries can be answered in $\mathcal{O}(\log^3 n)$ time.

Proof. The suffix trees $\mathcal{T}(X^R)$ and $\mathcal{T}(X)$ with heavy-path decompositions and ranges $\text{range}_{\mathcal{T}^R}$ and $\text{range}_{\mathcal{T}}$, respectively, take $\mathcal{O}(n)$ space and $\mathcal{O}(n)$ time to construct. The $\mathcal{O}(\log^2 n)$ weighted collections of points can be constructed in $\mathcal{O}(n \log^2 n)$ time by Lemma 6. Then the data structures for range maximum queries in 2-d (Lemma 3) in total take $\mathcal{O}(n \log^3 n)$ space and require $\mathcal{O}(n \log^4 n)$ expected time to construct.

To compute $\text{LCF}_2(i, \alpha)$, we perform the following steps. First, we compute the loci p and q in $\mathcal{O}(\log n \log \log n)$ time. Then, in $\mathcal{O}(\log n)$ time we compute the collections $H(p)$ and $H(q)$ of prefix fragments of heavy paths. Finally, for each pair $\pi'_1 \in H(p)$ and $\pi'_2 \in H(q)$, we answer range maximum queries of the form (1)–(9), each in $\mathcal{O}(\log n)$ time. This gives $\mathcal{O}(\log^3 n)$ total query time. \square

Lemmas 4 and 7 lead to the main result of this paper.

Theorem 8. LCF AFTER ONE SUBSTITUTION can be computed in $\mathcal{O}(\log^3 n)$ time, after $\mathcal{O}(n \log^4 n)$ expected time and $\mathcal{O}(n \log^3 n)$ space preprocessing.

Corollary 9. Given two strings S and T over a constant-sized alphabet, the answers to all $\Theta(n)$ possible LCF AFTER ONE SUBSTITUTION queries can be computed in $\mathcal{O}(n \log^4 n)$ expected time and $\mathcal{O}(n \log^3 n)$ space.

6 Conclusions

We have presented an $\tilde{\mathcal{O}}(n)$ -space data structure that can be constructed in $\tilde{\mathcal{O}}(n)$ expected time and supports $\tilde{\mathcal{O}}(1)$ -time computation of an LCF of two strings S and T , each of length at most n , over an integer alphabet after one letter substitution in S . Notably, our algorithm can be easily modified to work if we also allow for single letter insertions or deletions in S .

An open question is to extend this result to a fully dynamic case; that is, to propose a data structure that allows for *subsequent* edit operations in one or in both strings supporting fast computation of an LCF after each such operation.

References

1. Alstrup, S., Brodal, G.S., Rauhe, T.: New data structures for orthogonal range searching. In: FOCS, pp. 198–207. IEEE Computer Society (2000)
2. Amir, A., Landau, G.M., Lewenstein, M., Sokol, D.: Dynamic text and static pattern matching. ACM Trans. Algorithms **3**(2), 19 (2007)
3. Babenko, M.A., Starikovskaya, T.A.: Computing the longest common substring with one mismatch. Probl. Inf. Transm. **47**(1), 28–33 (2011)
4. Bender, M.A., Farach-Colton, M.: The LCA problem revisited. In: Gonnet, G.H., Viola, A. (eds.) LATIN 2000. LNCS, vol. 1776, pp. 88–94. Springer, Heidelberg (2000). doi:[10.1007/10719839_9](https://doi.org/10.1007/10719839_9)
5. Chi, L., Hui, K.: Color set size problem with applications to string matching. In: Apostolico, A., Crochemore, M., Galil, Z., Manber, U. (eds.) CPM 1992. LNCS, vol. 644, pp. 230–243. Springer, Heidelberg (1992). doi:[10.1007/3-540-56024-6_19](https://doi.org/10.1007/3-540-56024-6_19)

6. Crochemore, M., Hancart, C., Lecroq, T.: Algorithms on Strings. Cambridge University Press, Cambridge (2007)
7. Farach, M.: Optimal suffix tree construction with large alphabets. In: FOCS, pp. 137–143. IEEE Computer Society (1997)
8. Flouri, T., Giaquinta, E., Kobert, K., Ukkonen, E.: Longest common substrings with k mismatches. *Inf. Process. Lett.* **115**(6–8), 643–647 (2015)
9. Fredman, M.L., Komlós, J., Szemerédi, E.: Storing a sparse table with $O(1)$ worst case access time. *J. ACM* **31**(3), 538–544 (1984)
10. Grabowski, S.: A note on the longest common substring with k -mismatches problem. *Inf. Process. Lett.* **115**(6–8), 640–642 (2015)
11. Gusfield, D.: Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology. Cambridge University Press, New York (1997)
12. Kociumaka, T., Starikovskaya, T., Vildhøj, H.W.: Sublinear space algorithms for the longest common substring problem. In: Schulz, A.S., Wagner, D. (eds.) ESA 2014. LNCS, vol. 8737, pp. 605–617. Springer, Heidelberg (2014). doi:[10.1007/978-3-662-44777-2_50](https://doi.org/10.1007/978-3-662-44777-2_50)
13. Leimeister, C., Morgenstern, B.: kmacs: the k -mismatch average common substring approach to alignment-free sequence comparison. *Bioinformatics* **30**(14), 2000–2008 (2014)
14. Starikovskaya, T.: Longest common substring with approximately k mismatches. In: CPM. LIPIcs, vol. 54, pp. 21:1–21:11. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, Dagstuhl (2016)
15. Starikovskaya, T., Vildhøj, H.W.: Time-Space trade-offs for the longest common substring problem. In: Fischer, J., Sanders, P. (eds.) CPM 2013. LNCS, vol. 7922, pp. 223–234. Springer, Heidelberg (2013). doi:[10.1007/978-3-642-38905-4_22](https://doi.org/10.1007/978-3-642-38905-4_22)
16. Thankachan, S.V., Apostolico, A., Aluru, S.: A provably efficient algorithm for the k -mismatch average common substring problem. *J. Comput. Biol.* **23**(6), 472–482 (2016)
17. Thankachan, S.V., Chockalingam, S.P., Liu, Y., Apostolico, A., Aluru, S.: ALFRED: a practical method for alignment-free distance computation. *J. Comput. Biol.* **23**(6), 452–460 (2016)

String Processing and Information Retrieval
24th International Symposium, SPIRE 2017, Palermo,
Italy, September 26–29, 2017, Proceedings
Fici, G.; Sciortino, M.; Venturini, R. (Eds.)
2017, XIX, 318 p. 65 illus., Softcover
ISBN: 978-3-319-67427-8