Longest common substrings with k mismatches

Flouri, Tomas

2015


http://hdl.handle.net/10138/203401
https://doi.org/10.1016/j.ipl.2015.03.006

Downloaded from Helda, University of Helsinki institutional repository.

This is an electronic reprint of the original article.

This reprint may differ from the original in pagination and typographic detail.

Please cite the original version.
Longest common substrings with \( k \) mismatches

Tomas Flourí\(^ a \), Emanuele Giaquinta\(^ {b,*} \), Kassian Kobert\(^ a \), Esko Ukkonen\(^ c \)

\(^ a \) Heidelberg Institute for Theoretical Studies, Germany
\(^ b \) Department of Computer Science, Aalto University, Finland
\(^ c \) Department of Computer Science, University of Helsinki, Finland

A R T I C L E   I N F O

Article history:
Received 8 September 2014
Received in revised form 16 March 2015
Accepted 16 March 2015
Available online 20 March 2015
Communicated by L. Kowalik

Keywords:
Combinatorial problems
String algorithms
Hamming distance
Longest common substring

A B S T R A C T

The longest common substring with \( k \)-mismatches problem is to find, given two strings \( S_1 \) and \( S_2 \), a longest substring \( A_1 \) of \( S_1 \) and \( A_2 \) of \( S_2 \) such that the Hamming distance between \( A_1 \) and \( A_2 \) is \( \leq k \). We introduce a practical \( O(nm) \) time and \( O(1) \) space solution for this problem, where \( n \) and \( m \) are the lengths of \( S_1 \) and \( S_2 \), respectively. This algorithm can also be used to compute the matching statistics with \( k \)-mismatches of \( S_1 \) and \( S_2 \) in \( O(nm) \) time and \( O(m) \) space. Moreover, we also present a theoretical solution for the \( k = 1 \) case which runs in \( O(n \log m) \) time, assuming \( m \leq n \), and uses \( O(m) \) space, improving over the existing \( O(nm) \) time and \( O(m) \) space bound of Babenko and Starikovskaya [1].

© 2015 The Authors. Published by Elsevier B.V. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

1. Introduction

In this paper we study the longest common substring (or factor) with \( k \)-mismatches problem (\( k \)-LCF for short\(^ 1 \)) which consists in finding the longest common substring of two strings \( S_1 \) and \( S_2 \), while allowing for at most \( k \) mismatches, i.e., the Hamming distance between the two substrings is \( \leq k \). This problem is a generalization of the Longest Common Substring problem [2–4] and is similar to the threshold all-against-all problem defined by Gusfield [2] and to the local alignment problem of biological sequence analysis. In the threshold all-against-all problem the goal is to find all the pairs of substrings of \( S_1 \) and \( S_2 \) such that the corresponding edit distance is less than a given number \( d \). The difference in the \( k \)-LCF problem is that the distance used is the Hamming distance rather than the edit distance, and that we are interested in the pairs of substrings of maximal length only. In the local alignment problem, which can be solved in \( O(|S_1| \cdot |S_2|) \) time using the Smith–Waterman algorithm [5], the goal is to compute a pair of substrings of \( S_1 \) and \( S_2 \) such that the corresponding similarity, according to a suitable scoring function, is maximum over all the pairs of substrings. In particular, if the scoring function is such that the score of a match is 1, the score of a mismatch is 0 and gaps are not allowed, a solution of the local alignment problem is comparable to one of the \( k \)-LCF problem, with the difference that there is no bound on the number of mismatches.

Babenko and Starikovskaya [1] studied the case of 1 mismatch only and presented an algorithm for the 1-LCF problem which runs in \( O(|S_1| \cdot |S_2|) \) time. A closely related problem is the one of computing the matching statistics with \( k \) mismatches. The matching statistics, introduced by Chang and Lawler [6] for the approximate string matching problem, is an array \( ms \) of \( |S_2| \) integers such that \( ms[i] \) is the length of the longest substring of \( S_2 \) that starts at position \( i \) and matches exactly some substring of \( S_1 \), for \( i = 0, \ldots, |S_2| - 1 \). A natural generalization is obtained by allowing the matching to be approximate, with respect to the Hamming distance. Recently, Leimeister and Morgenstern [7] presented a greedy heuristic for the computation of the matching statistics with \( k \) mismatches, which runs

\(^ 1 \) We use the \( k \)-LCF abbreviation as LCS usually refers to the Longest Common Subsequence problem.
in $O(|S_2| \cdot k \cdot z)$ time, where $z$ is the maximum number of occurrences of $S_2$ in a string of maximal length which occurs in both $S_1$ and $S_2$.

In this paper we present two novel contributions. Our first result is an efficient algorithm for the $k$-LCF problem which runs in time $O(|S_1| \cdot |S_2|)$ and only requires a constant amount of space. This algorithm can also be used to compute the matching statistics with $k$ mismatches with no overhead in the time complexity, i.e., in $O(|S_1| \cdot |S_2|)$ time, and using $O(|S_2|)$ space. Our second result is an algorithm for the 1-LCF problem, i.e., for the $k=1$ case. We show how to solve this instance in a more time efficient manner by using results from Crochemore et al. [8] for finding the longest generalized repeat(s) with one block of $k$ adjacent don’t care symbols. Assuming $|S_2| \leq |S_1|$, our algorithm takes time $O(|S_1| \log |S_2|)$, improving over the previous bound of $O(|S_1| \cdot |S_2|)$.

2. Basic definitions

Let $\Sigma$ be a finite alphabet of symbols and let $\Sigma^*$ be the set of strings over $\Sigma$. Given a string $S \in \Sigma^*$, we denote by $|S|$ the length of $S$ and by $S[i]$ the $i$-th symbol of $S$, for $0 \leq i < |S|$. Given two strings $S$ and $S'$, $S'$ is a substring of $S$ if there are indices $0 \leq i < j < |S|$ such that $S'[i]...S[j]$. If $i = 0$ ($j = |S| - 1$) then $S'$ is a prefix (suffix) of $S$. We denote by $S[i,j]$ the substring of $S$ starting at position $i$ and ending at position $j$. For $i > j$ we obtain the empty string $\varepsilon$. Finally, we denote by $S^r = S[|S| - 1]...S[0]$ the reverse of the string $S$.

The suffix tree $T(S)$ of a string $S$ is a rooted directed tree with $|S|$ leaves and edge labels over $(\Sigma \cup \{\varepsilon\})^*$, where $S \notin \Sigma$ and $S' \in \Sigma^*$. Each internal node has at least two children and is such that the edge labels of the children have different first symbols. For each leaf $i$, the concatenation of the edge labels on the path from the root to leaf $i$ is equal to $S'[i..|S'| - 1]$. Assuming a constant size alphabet, the suffix tree can be built in $O(|S|)$ time [2]. For any node $u$ in $T(S)$, depth$(u)$ denotes the length of the string labeling the path from the root to $u$. For any pair of nodes $u, v$ in $T(S)$, LCA($u, v$) denotes the lowest common ancestor of $u$ and $v$, i.e., the deepest node in $T(S)$ that is ancestor of both $u$ and $v$. The suffix tree can be preprocessed in $O(|S|)$ time so as to answer LCA queries in constant time [9]. We denote by $B(S)$ the binary suffix tree obtained by replacing each node $u$ in $T(S)$ with out-degree at least 2 with a binary tree with $d - 1$ internal nodes whose depth values are equal to depth$(u)$ and $d - 2$ internal edges, where the $d$ leaves are the $d$ children of $u$. The binary suffix tree can be built in $O(|S|)$ time [8]. The generalized suffix tree $T(S_1, S_2)$ of two strings $S_1$ and $S_2$ is the suffix tree built over $S' = S_1S_2$, where $S_1, S_2 \notin \Sigma$, such that the leaves are numbered with a pair $(s$-index) and for each leaf $(j, l)$ the concatenation of the edge labels on the path from the root to the leaf is equal to $S[j..|S| - 1]S[l..|S| - 1]$. The index of a leaf $(j, l)$ is the starting position of $S[j..|S| - 1]S[l..|S| - 1]$ in $S_1S_2$. We use the notation $B(S_1, S_2)$ to denote the binary generalized suffix tree of $S_1$ and $S_2$.

3. The longest common substring with $k$ mismatches problem

Let $S_1$ and $S_2$ be two strings with $n = |S_1|, m = |S_2|$. W.l.o.g. we assume that $n \geq m$. Given an integer $k$, let $\phi(i, j)$ be the length of the longest substring of $S_1$ and $S_2$ ending at position $i$ and $j$, respectively, such that the two substrings have Hamming distance at most $k$. Formally, $\phi(i, j)$ is equal to the largest integer $l \leq \min(i, j) + 1$ such that $|[0 \leq h \leq l - 1 | S_1[i - h] \neq S_2[j - h]| \leq k,$ for $0 \leq i < n, 0 \leq j < m$. The longest common substring with $k$-mismatches problem consists in, given two strings $S_1$ and $S_2$ and an integer $k$, finding the length of the longest substrings of $S_1$ and $S_2$ with Hamming distance at most $k$, i.e., $\text{max}_{i,j} \phi(i, j)$.

4. A practical algorithm for arbitrary $k$

In this section we present a practical algorithm for the $k$-LCF problem. By definition, $\phi(i, j)$ is also the length of the longest suffixes of $S_1[0..i]$ and $S_2[0..j]$ with Hamming distance at most $k$. Our algorithm computes all the values $\phi(i, j)$ based on this alternative formulation. The idea is to iterate over the $\phi$ matrix diagonal-wise and compute, for a fixed $(i, j) \in \{(0, 0), (0, 1), \ldots, (0, m - 1)\} \cup \{(1, 0), (2, 0), \ldots, (n - 1, 0)\}$, the values $\phi(i + p, j + p)$, for $0 \leq p < \min(n - i, m - j)$, i.e., the diagonal starting at $(i, j)$, in $O(m)$ time. Let $Q$ be an (empty) queue data structure and $s = 0$, for a given pair $(i, j)$. The algorithm iterates over $p$ maintaining the invariant that $p - s$ is the length of the longest common suffix of $S_1[i..i + p - 1]$ and $S_2[j..j + p - 1]$ up to $k$-mismatches, i.e., $p - s = \phi(i + p, j + p)$, and that $Q$ contains exactly the positions in $S_1$ of the mismatches between $S_1[i..i + p - 1]$ and $S_2[j..j + p - 1]$ with the order of elements in the queue matching their natural order.

At the beginning the invariant holds since $Q$ is empty, $p - s = 0$ and $S_1[i + s..i + p - 1] = S_2[j + s..j + p - 1] = \varepsilon$. Suppose that the invariant holds up to position $p$. If $S_1[i + p] = S_2[j + p]$ then the invariant trivially holds also for $p + 1$ with $s' = s$ and $Q' = Q$. Otherwise, we have a mismatch between $S_1[i + p]$ and $S_2[j + p]$. If $|Q| < k$, then the invariant also holds for $p + 1$ with $s' = s$ and $Q'$ equal to $Q$ after an ENQUEUE($Q$, $p$) operation. Instead, if $|Q| = k$, the pair of suffixes $S_1[i + r..i + p]$ and $S_2[j + r..j + p]$, for $r = \ldots$ min $Q$, match with $k + 1$ mismatches and $r = \min Q + 1$ is the minimum position for which the corresponding suffixes match with $k$ mismatches. Hence, in this case the invariant also holds for $p + 1$ with $s' = \min Q + 1$ and $Q'$ equal to $Q$ after a DEQUEUE operation followed by an ENQUEUE($Q$, $p$) operation.

The algorithm maintains the largest length found up to the current iteration and the starting positions of the corresponding substrings in $S_1$ and $S_2$, such that the position in $S_1$ is minimal, in three integers $\ell, r_1$, and $r_2$. Each time $p - s > \ell$ it updates their values accordingly. The code of the algorithm is shown in Fig. 1. The time complexity of one iteration of the algorithm is $O(1)$ if the queue operations take constant time, which yields $O(m)$ time for a
fixed i and O(nm) time in total. The space complexity is O(k), as the queue contains at most k elements at any iteration.

For scanning one diagonal of φ, the algorithm needs time that is proportional to the length of the diagonal. This can be improved such that the time requirement becomes proportional to the number of mismatches along the diagonal, by using the well-known technique that performs LCA queries on the generalized suffix tree of S1 and S2 to find, in constant time, how far the next mismatch is from the current one [10]. This gives an algorithm for the k-LCF problem that runs in time proportional to the number of pairs (i, j) such that S1[i] ≠ S2[j].

Constant-space variant the algorithm can also be modified to use O(1) space at the price of a constant factor in the running time. We replace the queue with one integer q, encoding the number of mismatches (number of elements in the queue). The DEQUEUE and ENQUEUE operations then become q ← q − 1 and q ← q + 1, respectively. The update of s requires the computation of min Q + 1, which, by definition, is equal to the smallest position s' > s such that S1[i + s' − 1] ≠ S2[j + s' − 1]. To this end, we simply scan S1 and S2 from position i + s and j + s, respectively, until we find a mismatch. As each symbol of S1 and S2 is looked up at most twice, the time complexity does not change. In practice, using an explicit queue is preferable, as it allows one to avoid rescanning the already scanned parts of the strings.

Matching statistics with k mismatches finally, we describe how to compute the matching statistics with k mismatches of S2 with respect to S1. The matching statistics with k mismatches of S2 w.r.t. S1 is an array ms[k] of m integers such that ms[k][j] is the length of the longest prefix of S2[i..m − 1] that matches a substring of S1, with at most k mismatches, for i = 0, . . . , m − 1. Using the algorithm described above, the array ms[k] can be computed in O(nm) time as follows: first, we initialize each slot of ms[k] to 0; then, we run our algorithm on $S_1'$ and $S_2'$, i.e., on the reverse of the strings S1 and S2, and for each computed cell φ(i, j) we set ms[k][m − 1 − j] = max(ms[k][m − 1 − j], φ(i, j)). At the end of the procedure we thus have ms[k][m − 1 − j] = max(φ(i, j)), for 0 ≤ j < m. The correctness of this procedure follows by observing that i) a suffix S'[0..i] is the reverse of a prefix of S[S1 − 1 − i..S1], for any string S and 0 ≤ i < |S|, and ii) φ(i, j) is the length of the longest suffixes of S'[0..i] and S'[0..j] with Hamming distance at most k. Hence, maxS[k][j] is the length of the longest prefix of S[m − 1 − j..m − 1] that matches a substring of S1 with at most k mismatches.

Note that the φ matrix for S1 and S2 immediately gives a dual matching statistics, where ms[k][i] is defined as the length of the longest suffix of S2[0..i] that matches a substring of S1 with at most k mismatches. In practical applications this alternative matching statistics could be equally good.

5. Longest common substring with 1 mismatch

In this section we describe an algorithm that solves the 1-LCF problem. We first introduce some necessary technical definitions. Given a string S, a pair of substrings ((p1, q1), (p2, q2)) of S is a repeated pair if S[p1..q1] = S[p2..q2]. A repeated pair ((p1, q1), (p2, q2)) is left-maximal (right-maximal) if S[p1−1] ≠ S[p2−1] (S[q1+1] ≠ S[q2+1]). Given a string S, a repeat is a substring of S that corresponds to a repeated pair. A repeat w of S is left-maximal (right-maximal) if there exists a left-maximal (right-maximal) repeated pair ((p1, q1), (p2, q2)) such that S[p1+1] = S[p2+1] = w. Let * be the don’t care symbol, i.e., a symbol that matches any symbol of Σ. A k-repeat of S is a string of the form u*kv that matches more than one substring of S, where u, v ∈ Σ*. A longest k-repeat is a k-repeat of maximum length. A necessary condition for a k-repeat u*kv to be longest is that, for each pair ((p1, q1), (p2, q2)) of substrings matching the repeat, ((p1, p1 + |u| − 1), (p2, p2 + |u| − 1)) is a left-maximal repeated pair and ((p1 + |u| + k, q1), (p2 + |u| + k, q2)) is a right-maximal repeated pair.

The idea is to reduce the 1-LCF problem to the one of computing the longest 1-repeats of Ŝ = S1S1S2S2 that occur in both S1 and S2, where S1, S2 are two symbols not in Σ. Let ℓ = maxS[1]i φ(i, j) for k = 1, and let ℓ, j be such that φ(i, j') = ℓ. Consider the strings A1 = S1[1..ℓ − 1..1, i′] and A2 = S2[j′..ℓ − 1..j′]. It is not hard to see that the string A1[0..p − 1] + A2[p + 1..ℓ − 1] is a longest 1-repeat of Ŝ that occurs in both S1 and S2, where either A1 = A2 and 0 ≤ p ≥ ℓ − 1 or A1 ≠ A2 and p is the position corresponding to the single mismatch between A1 and A2.

To this end, we use a modified version of the algorithm ALL-LENGTHS-K-REPEATS by Crochemore et al. to find the longest k-repeats of a string [8]. The idea is to run this algorithm on the string Ŝ with k = 1. With this input, the original algorithm reports all the longest 1-repeats of Ŝ. To solve our problem we need to add the constraint that the 1-repeats must occur in both S1 and S2. As the longest such repeats can be shorter than the unconstrained longest 1-repeats of Ŝ, the ALL-LENGTHS-K-REPEATS algorithm must be modified accordingly.

The ALL-LENGTHS-K-REPEATS algorithm is structured in the following steps:

---

**Algorithm:**

1. **k-LCF(S1, S2, k):**
   1. n ← |S1|
   2. m ← |S2|
   3. r1 ← 0, r2 ← 0
   4. **for** d ← m − 1 **to** n − 1 **do**
   5. i ← max(d − 0, 0) + d
   6. j ← max(d, 0)
   7. Q ← φ
   8. s ← 0, p ← 0
   9. **while** p ≤ min(n − i, m − j) − 1 **do**
   10. if S[i + p] ≠ S[j + p] **then**
   11. if Q = k **then**
   12. s ← min Q + 1
   13. DEQUEUE(Q)
   14. ENQUEUE(Q, p)
   15. p ← p + 1
   16. if p − s > ℓ **then**
   17. ℓ ← p − s
   18. r1 ← i + s
   19. r2 ← j + s

**Fig. 1.** The algorithm to compute the longest common substring up to k-mismatches of two strings.
1. build the suffix tree $T(S)$ of $S$ and compute the ordering of the leaves induced by a depth-first visit; build the binary suffix tree $B(S')$ of $S'$ and associate to each leaf $u$ with index $i$ a list $A_u$ equal to $(\text{no}(i))$, if $i \geq k$, and to $\emptyset$ otherwise, where $i = |S| - i + k$; $\gamma \leftarrow 0$

2. for $u \in B(S')$ in depth-first order with children $u_1$ and $u_2$ do
   (a) $\text{FIND-LONGEST}(A_{u_1}, A_{u_2}, \text{depth}(u) + k, \gamma)$
   (b) $A_u \leftarrow \text{MERGE}(A_{u_1}, A_{u_2})$

where $\text{MERGE}(L_1, L_2)$ merges two lists $L_1, L_2$, and $\text{FIND-}\text{LONGEST}$ is defined as follows:

\begin{align*}
\text{FIND-}\text{LONGEST}(L_1, L_2, l, \gamma) & \quad 1. \quad (i_1, i_2) \leftarrow (1,2) \\
& \quad 2. \text{if } |L_1| \geq |L_2| \text{ then } (i_1, i_2) \leftarrow (2,1) \\
& \quad 3. \text{for } p \in L_{i_1} \text{ in ascending order do} \\
& \quad \quad q \leftarrow \max\{j \in L_{i_2} \mid j \leq p\}, r \leftarrow \min\{j \in L_{i_2} \mid j > p\} \\
& \quad \quad \upsilon_{pq} = \text{LCA}(\text{no}^{-1}(p), \text{no}^{-1}(q)), \upsilon_{pr} = \text{LCA}(\text{no}^{-1}(p), \text{no}^{-1}(r)) \\
& \quad \quad \gamma \leftarrow \max(\gamma, l + \max(\text{depth}(\upsilon_{pq}), \text{depth}(\upsilon_{pr})))
\end{align*}

where the LCA queries are performed on $T(S)$. At the end of the algorithm the value of $\gamma$ is the length of the longest $k$-repeat(s) of $S$. If the lists $L_1, L_2$ are implemented using AVL-trees, the time complexity of the merge and $\text{FIND-}\text{LONGEST}$ procedures is $O(m \log(n/m))$ [11], where $m = \min(|L_1|, |L_2|), n = \max(|L_1|, |L_2|)$, and the algorithm can be proved to run in $O(|S| \log |S|)$ time. The main property on which the algorithm is based is the following lemma:

**Lemma 1.** Let $u, v, w$ be leaves in the suffix tree of $S$ with corresponding depth-first ordering of leaves no. If $\text{no}(u) < \text{no}(v) < \text{no}(w)$ or $\text{no}(w) < \text{no}(v) < \text{no}(u)$ then depth($\text{LCA}(u, v)$) $\geq$ $\text{depth}(\text{LCA}(u, w))$.

Let $L(u)$ be the list containing the integer $i$ for each leaf with index $i$ in the subtree of node $u$ of $B(S')$. The idea is to iterate over all the left-maximal repeats of $S$ using $B(S')$ and for each pair $(p_1, p_2)$ of indexes in $L(u)$ of such a repeat $u$ compute the right-maximal repeat starting at position $p_1$ and $p_2$ using an LCA query on $T(S)$. It turns out, by the above lemma, that, for a given index $p \in L(u)$, it is enough to check the pairs $(p, q)$ and $(p, r)$ where $q$ and $r$ are the indexes of the closest leaves to leaf $p$ in $T(S)$, with respect to the ordering no, such that $q, r \in L(u)$.

Our modification consists in the following: we replace $T(S)$ with the generalized suffix tree of $S_1$ and $S_2$ and $B(S')$ with the binary generalized suffix tree of $S_1'$ and $S_2'$. Let $L_1(u)$ be a list containing the integer $i$, for each leaf with $s$-index $(j, l)$ and index $i$ in the subtree of node $u$ of $B(S_1'(S_2'))$, provided that $l \geq k$, for $j = 1, 2$. The condition $l \geq k$ ensures that the occurrence of $u$ in $S_1$ ending at position $|S_1| - 1$ can be extended by $k$ don’t care symbols to the right, as otherwise there can be no $k$-repeat with left part equal to the reverse of $u$ label matching a prefix of $S_1'[S_1] - 1 - \text{depth}(u) + 1, |S_1| - 1 |$. Our goal is to iterate over pairs in $L_1(u) \times L_2(u)$ only by computing, for a given index $p \in L_1(u)$, the indexes $q$ and $r$ of the closest leaves to leaf $p$ in $T(S_1, S_2)$, with respect to the ordering no, such that $q, r \in L_2(u)$, and vice versa if $p \in L_2(u)$. To accomplish this, it is enough to associate to each leaf $u$ of $B(S_1', S_2')$ with $s$-index $(j, l)$ and index $i$ two lists, $A_u^1$ and $A_u^2$: if $l < k$ the lists are empty; otherwise, if $j = 1$ then $A_u^1 = \{\text{no}(i)\}$ and $A_u^2 = \emptyset$, and vice versa if $j = 2$. Then, we change the operations in the second step of the algorithm as follows:

\begin{align*}
(a) \quad & \text{FIND-}\text{LONGEST}(A_{u_1}^1, A_{u_2}^2, \text{depth}(u) + k, \gamma) \\
(b) \quad & \text{FIND-}\text{LONGEST}(A_{u_1}^2, A_{u_2}^1, \text{depth}(u) + k, \gamma) \\
(c) \quad & A_u^1 \leftarrow \text{MERGE}(A_{u_1}^1, A_{u_2}^1) \\
(d) \quad & A_u^2 \leftarrow \text{MERGE}(A_{u_1}^2, A_{u_2}^2)
\end{align*}

In this way we iterate only over pairs $((p_1, q_1), (p_2, q_2))$ of $S$ matching a 1-repeat and such that $0 \leq p_1, q_1 \leq |S_1| - 1$ and $|S_1| + 1 \leq p_2, q_2 \leq |S_1| + |S_2|$, or vice versa. At the end of the algorithm the value of $\gamma$ is the length of the longest $k$-repeat(s) of $S$ that occur in both $S_1$ and $S_2$.

We now prove that the time complexity of steps a, b, c, and d is $O(m \log(n/m))$, where $m = \min(|A_u^1|, |A_u^2|), n = \max(|A_u^1|, |A_u^2|)$, i.e., there is only a constant overhead compared to the original algorithm. Suppose w.l.o.g. that $m = |A_u^1|, n = |A_u^2|$ and let $m_1 = |A_u^1|$ and $n_1 = |A_u^2|$, for $1 \leq i, j \leq 2$. Note that $m > m_1 + m_2$ and $n > n_1 + n_2$. Step a, b, c, or d takes i) $O(m \log(n_1/m_1))$ time, if $m_1 < n_2$; ii) $O(n_1 \log(n_1/m_1))$ time otherwise, where $n_2 \leq m_1$. We show that $\log(n_1/m_1) \leq \log(m/m_1)$ for any $1 \leq a \leq m_1$. This inequality can be written as $\frac{\log(n_1/m_1)}{\log(m/m_1)} \leq \log n$. We have $f(x) = x \log x$ and, by the mean value theorem, there exists $c \in (a, m)$ such that $\frac{f(m) - f(a)}{m - a} = \log c \leq \log m \leq \log n$.

The total time complexity of our algorithm for the 1-LCF problem is thus $O((n + m) \log(n + m))$, Assuming $m \leq n$, we can reduce it to $O(n \log m)$ by partitioning $S_1$ into overlapping substrings of length $2m$ such that the overlap between two consecutive substrings is of length $m$, and running the algorithm on each substring and $S_2$. Formally, we run the algorithm on $S_1[m \cdots \min(m + 2m, n) - 1]$ and $S_2$ and obtain a value $\gamma_1$, for $0 \leq i < |n/m|$. Then, $\gamma = \max_i \gamma_i$. The time complexity of this algorithm is $O((n/m) \log(m)) = O(n \log m)$.

**Acknowledgements**

We thank the anonymous reviewers for helpful comments.

**References**


