Time-Space Trade-Offs for the Longest Common Substring Problem

Tatiana Starikovskaya[1] and Hjalte Wedel Vildhøj[2]

[1] Moscow State University, Department of Mechanics and Mathematics, tat.starikovskaya@gmail.com
[2] Technical University of Denmark, DTU Compute, hwv@hwv.dk

Abstract. The Longest Common Substring problem is to compute the longest substring which occurs in at least $d \geq 2$ of m strings of total length n. In this paper we ask the question whether this problem allows a deterministic time-space trade-off using $O(n^{1+\varepsilon})$ time and $O(n^{1-\varepsilon})$ space for $0 \leq \varepsilon \leq 1$. We give a positive answer in the case of two strings ($d = m = 2$) and $0 < \varepsilon \leq 1/3$. In the general case where $2 \leq d \leq m$, we show that the problem can be solved in $O(n^{1-\varepsilon})$ space and $O(n^{1+\varepsilon} \log^2 n(d \log^2 n + d^2))$ time for any $0 \leq \varepsilon < 1/3$.

1 Introduction

The *Longest Common Substring (LCS) Problem* is among the fundamental and classic problems in combinatorial pattern matching [6]. Given two strings T_1 and T_2 of total length n, this is the problem of finding the longest substring that occurs in both strings. In 1970 Knuth conjectured that it was not possible to solve the problem in linear time [10], but today it is well-known that the LCS can be found in $O(n)$ time by constructing and traversing a suffix tree for T_1 and T_2 [6]. However, obtaining linear time comes at the cost of using $\Theta(n)$ space, which in real-world applications might be infeasible.

In this paper we explore solutions to the LCS problem that achieve sublinear, i.e., $o(n)$, space usage[3] at the expense of using superlinear time. For example, our results imply that the LCS of two strings can be found deterministically in $O(n^{4/3})$ time while using only $O(n^{2/3})$ space. We will also study the time-space trade-offs for the more general version of the LCS problem, where we are given m strings $T_1, T_2, \ldots, T_m$ of total length n, and the goal is to find the longest common substring that occurs in at least d of these strings, $2 \leq d \leq m$.

1.1 Known Solutions

For $m = d = 2$ the LCS is the longest common prefix between any pair of suffixes from T_1 and T_2. Naively comparing all pairs leads to an $O(n^2|\text{LCS}|)$ time and $O(1)$ space solution, where $|\text{LCS}|$ denotes the length of the LCS. As

[3] We assume the input is in read-only memory and not counted in the space usage.

already mentioned we can also find the LCS in $O(n)$ time and space by finding the deepest node in the suffix tree that has a suffix from both T_1 and T_2 in its subtree. Alternatively, we can build a data structure that for any pair of suffixes can be queried for the value of their longest common prefix. Building such a data structure is known as the *Longest Common Extension (LCE) Problem* and it has several known solutions [2,7]. If a data structure for a string of length n with query time $q(n)$ and space usage $s(n)$ can be built in time $p(n)$, then this implies a solution for the LCS problem using $O(q(n)n^2 + p(n))$ time and $O(s(n))$ space. For example using the deterministic data structure of Bille et al. [2], the LCS problem can be solved in $O(n^{2(1+\varepsilon)})$ time and $O(n^{1-\varepsilon})$ space for any $0 \leq \varepsilon \leq 1/2$.

In the general case where $2 \leq d \leq m$, the LCS can still be found in $O(n)$ time and space using the suffix tree approach. Using Rabin-Karp fingerprints [9] we can also obtain an efficient randomised algorithm using sublinear space. The algorithm is based on the following useful trick: Suppose that we have an efficient algorithm for deciding if there is a substring of length i that occurs in at least d of the m strings. Moreover, assume that the algorithm outputs such a string of length i if it exists. Then we can find the LCS by repeating the algorithm $O(\log |\text{LCS}|)$ times in an exponential search for the maximum value of i. To determine if there is a substring of length i that occurs in at least d strings, we start by checking if any of the $n^{1-\varepsilon}$ first substrings of length i occurs at least d times. We can check this efficiently by storing their fingerprints in a hash table and sliding a window of length i over the strings T_j, $j = 1, \ldots, m$. For each substring we look up its fingerprint in the hash table and increment an associated counter if it is the first time we see this fingerprint in T_j. If at any time a counter exceeds d, we stop and output the window. In this way we can check all i length substrings in $O(n^{\varepsilon})$ rounds each taking time $O(n)$. Thus, this gives a Monte Carlo algorithm for the general LCS problem using $O(n^{1+\varepsilon} \log |\text{LCS}|)$

	Space	**Time**	**Trade-Off Interval**	**Description**
$d = m = 2$	$O(1)$	$O(n^2\|\text{LCS}\|)$		Naive solution.
	$O(n^{1-\varepsilon})$	$O(n^{2(1+\varepsilon)})$	$0 \leq \varepsilon \leq \frac{1}{2}$	Deterministic LCE d.s. [2]
	$O(n^{1-\varepsilon})$	$O(n^{2+\varepsilon} \log \|\text{LCS}\|)$ w.h.p.	$0 \leq \varepsilon \leq 1$	Randomised LCE d.s. [2]
	$O(n^{1-\varepsilon})$	$O(n^{1+\varepsilon})$	$0 < \varepsilon \leq \frac{1}{3}$	Our solution, $d{=}m{=}2$.
$2 \leq d \leq m$	$O(n^{1-\varepsilon})$	$O(n^{1+\varepsilon} \log \|\text{LCS}\|)$	$0 \leq \varepsilon \leq 1$	Randomised fingerprints. Correct w.h.p.
	$O(n^{1-\varepsilon})$	$O(n^{1+\varepsilon} \log^2 n(d \log^2 n + d^2))$	$0 \leq \varepsilon < \frac{1}{3}$	Our solution, $2{\leq}d{\leq}m$.
	$O(n)$	$O(n)$		Suffix tree.

Table 1. The first half summarises solutions for $d = m = 2$, and the second half summarises solutions for the general case. The complexity bounds are worst-case unless otherwise stated; w.h.p. means with probability at least $1 - 1/n^c$ for any constant c.

time and $O(n^{1-\varepsilon})$ space for all $0 \leq \varepsilon \leq 1$. From the properties of fingerprinting we know that the algorithm succeeds with high probability. The algorithm can also be turned into a Las Vegas algorithm by verifying that the fingerprinting function is collision free in $O(n^2)$ time. Table 1 summarises the solutions.

1.2 Our Results

We show the following main result:

Theorem 1. *Given m strings $T_1, T_2, \ldots, T_m$ of total length n, an integer $2 \leq d \leq m$ and a trade-off parameter ε, the longest common substring that occurs in at least d of the m strings can be found in*

(i) $O(n^{1-\varepsilon})$ space and $O(n^{1+\varepsilon})$ time for $d=m=2$ and $0<\varepsilon\leq\frac{1}{3}$, or in

(ii) $O(n^{1-\varepsilon})$ space and $O(n^{1+\varepsilon}\log^2 n(d\log^2 n+d^2))$ time for $2\leq d\leq m$, $0\leq\varepsilon<\frac{1}{3}$.

The main innovation in these results is that they are both deterministic. Moreover, our first solution improves over the randomised fingerprinting trade-off by removing the $\log|\text{LCS}|$ factor. The basis of both solutions is a sparse suffix array determining the lexicographic order on $O(n^{1-\varepsilon})$ suffixes sampled from the strings $T_1, T_2, \ldots, T_m$ using difference covers.

2 Preliminaries

Throughout the paper all logarithms are base 2, and positions in strings are numbered from 1. Notation $T[i..j]$ stands for a substring $T[i]T[i+1]\cdots T[j]$ of T, and $T[i..]$ denotes the suffix of T starting at position i. The longest common prefix of strings T_1 and T_2 is denoted by $\text{lcp}(T_1, T_2)$.

2.1 Suffix Trees

We assume a basic knowledge of *suffix trees*. In order to traverse and construct suffix trees in linear time and space, we will assume that the size of the alphabet is constant. Thus, the suffix tree for a set of strings $\mathcal{S}$, denoted $ST(\mathcal{S})$, together with suffix links, can be built in $O(n)$ time and space, where n is the total length of strings in $\mathcal{S}$ [6]. We remind that a suffix link of a node labelled by a string ℓ points to the node labelled by $\ell[2..]$ and that suffix links exist for all inner nodes of a suffix tree. We need the following lemma:

Lemma 1. *Let $ST(\mathcal{S})$ be the suffix tree for a set of strings $\mathcal{S}$, and $\mathcal{A}$ be a set of all nodes (explicit or implicit) of $ST(\mathcal{S})$ labelled by substrings of another string T. I.e., the labels of the nodes in $\mathcal{A}$ are exactly all common substrings of T and strings from $\mathcal{S}$. Then $ST(\mathcal{S})$ can be traversed in $O(|T|)$ time so that*

(i) All nodes visited during the traversal will belong to $\mathcal{A}$, and
(ii) Every node in $\mathcal{A}$ will have at least one visited descendant.

Proof. We first explain how the tree is traversed. We traverse $ST(\mathcal{S})$ with T starting at the root. If a mismatch occurs or the end of T is reached at a node v (either explicit or implicit) labelled by a string ℓ we first jump to a node v' labelled by $\ell[2..]$. We do that in three steps: 1) walk up to the higher end u of the edge v belongs to; 2) follow the suffix link from u to a node u'; 3) descend from u' to v' comparing only the first characters of the labels of the edges with the corresponding characters of $\ell[2..]$ in $O(1)$ time. Then we proceed the traversal from the position of T at which the mismatch occurred. The traversal will end at the root of the suffix tree.

All nodes visited during the traversal are labelled by substrings of T, and thus belong to $\mathcal{A}$. For each i the traversal visits the deepest node of $ST(\mathcal{S})$ labelled by a prefix of $T[i..]$. Hence, conditions (i) and (ii) of the lemma hold. We now estimate the running time. Obviously, the number of successful matches is no more than $|T|$. We estimate the number of operations made due to unsuccessful matches by amortised analysis. During the traversal we follow at most $|T|$ suffix links and each time the depth of the current node decreases by at most one [6]. Hence, the number of up-walks is also bounded by $|T|$. Each up-walk decreases the current node-depth by one as well. On the contrary, traversal of an edge at step 3) increases the current node-depth by one. Since the maximal depth of a node visited by the traversal is at most $|T|$, the total number of down-walks is $O(|T|)$. □

2.2 Difference Cover Sparse Suffix Arrays

A *difference cover modulo* τ is a set of integers $DC_\tau \subseteq \{0, 1, \ldots, \tau - 1\}$ which for any i, j contains two elements i', j' such that $j - i \equiv j' - i' \pmod{\tau}$. For any τ a difference cover DC_τ of size at most $\sqrt{1.5\tau} + 6$ can be computed in $O(\sqrt{\tau})$ time [4]. Note that this size is optimal to within constant factors, since any difference cover modulo τ must contain at least $\sqrt{\tau}$ elements.

For a string T of length n and a fixed difference cover modulo τ, DC_τ, we define a *difference cover sample* $DC_\tau(T)$ as the subset of T's positions that are in the difference cover modulo τ, i.e.,

$$DC_\tau(T) = \{i \mid 1 \leq i \leq n \wedge \ i \bmod \tau \in DC_\tau\} .$$

The following lemma captures two important properties of difference cover samples that we will use throughout the paper. The proof follows immediately from the above definitions.

Lemma 2. *The size of $DC_\tau(T)$ is $O(n/\sqrt{\tau})$, and for any pair p_1, p_2 of positions in T there is an integer $0 \leq i < \tau$ such that both $(p_1 + i)$ and $(p_2 + i)$ are in $DC_\tau(T)$.*

We will consider difference cover samples of the string $T = T_1\$_1T_2\$_2 \cdots T_k\$_k$, i.e., the string obtained by concatenating and delimiting the input strings with unique characters $\$_1, \ldots, \$_k$. See Figure 1 for an example of a difference cover sample of two input strings.

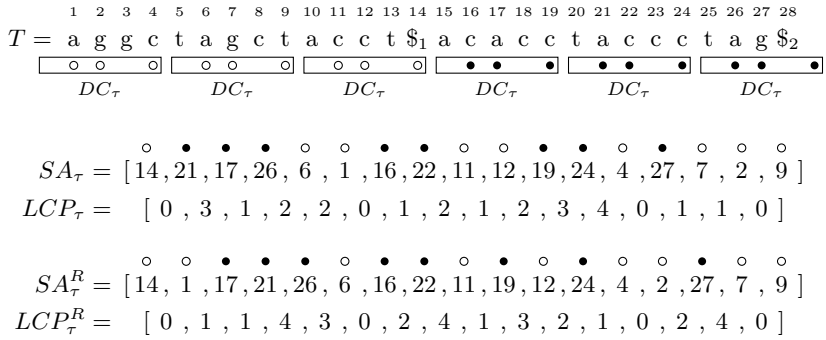

Fig. 1. The string $T = T_1\$_1T_2\$_2 = \texttt{aggctagctacct\$}_1\texttt{acacctaccctag\$}_2$ sampled with the difference cover $DC_\tau = \{1, 2, 4\}$ modulo 5. The resulting difference cover sample is $DC_\tau(T) = \{1, 2, 4, 6, 7, 9, 11, 12, 14, 16, 17, 19, 21, 22, 24, 26, 27\}$. Below the arrays SA_τ, LCP_τ, SA^R_τ and LCP^R_τ are shown. Sampled positions in T_1 and T_2 are marked by white and black dots, respectively.

The *difference cover sparse suffix array*, denoted SA_τ is the suffix array restricted to the positions of T sampled by the difference cover, i.e., it is an array of length $n/\sqrt{\tau}$ containing the positions of the sampled suffixes, sorted lexicographically. Similarly, we define the *difference cover sparse lcp array*, denoted LCP_τ, as the array storing the longest common prefix (lcp) values of neighbouring suffixes in SA_τ. Moreover, for a sampled position $p \in DC_\tau(T)$ we denote by $RB(p)$ the reversed substring of length τ ending in p, i.e., $RB(p) = T[p]T[p-1]\ldots T[p-\tau+1]$, and we refer to this string as the *reversed block* ending in p. As for the sampled suffixes, we define arrays SA^R_τ and LCP^R_τ for the reversed blocks. The first contains the sampled positions sorted according to the lexicographic ordering of the reversed blocks, and the latter stores the corresponding longest common prefix values. See Figure 1 for an example of the arrays SA_τ, LCP_τ, SA^R_τ and LCP^R_τ.

The four arrays can be constructed in $O(n\sqrt{\tau} + (n/\sqrt{\tau})\log(n/\sqrt{\tau}))$ time and $O(n/\sqrt{\tau})$ space [3,11]. To be able to compute the longest common prefix between pairs of sampled suffixes and pairs of reversed blocks in constant time, we use the well-known technique of constructing a linear space *range minimum query* data structure [5,1] for the arrays LCP_τ and LCP^R_τ.

3 Longest Common Substring of Two Strings

In this section we prove Theorem 1(i). We do so by providing two algorithms both using $O(n/\sqrt{\tau})$ space which are then combined to obtain the desired trade-off. The first one correctly computes the LCS if it has length at least τ, while the second one works if the length of the LCS is less than τ. In the second algorithm we must assume that $\tau \leq n^{2/3}$, which translates into the $\varepsilon \leq 1/3$ bound on the trade-off interval.

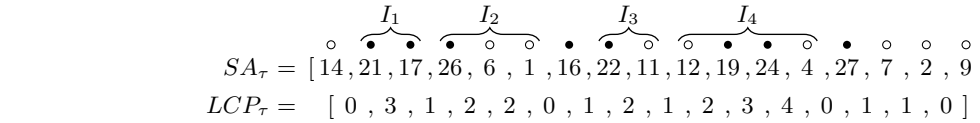

Fig. 2. The intervals of SA_τ containing the pairs with lcp values at least $\ell = 2$ for the string shown in Figure 1. The pair maximising the lcp value of the corresponding reversed blocks is $p'_1 = 11, p'_2 = 22$, which happens to be the LCS of T_1 and T_2: `ctacc`.

3.1 A Solution for Long LCS

We first compute a difference cover sample with parameter τ for the string $T = T_1\$_1T_2\$_2$, where $\$_1, \$_2$ are special characters that do not occur in T_1 or T_2. We then construct the arrays and the range minimum query data structures described in Section 2.2 for computing longest common prefixes between pairs of sampled suffixes or pairs of reversed blocks in constant time.

The LCS is the longest common prefix of suffixes $T[p_1..]$ and $T[p_2..]$ for some $p_1 \leq |T_1|$ and $p_2 > |T_1| + 1$. If $|\text{LCS}| \geq \tau$ then from the property of difference cover samples (Lemma 2) it follows that there is an integer $r < \tau$ such that $p'_1 = p_1 + r$ and $p'_2 = p_2 + r$ are both in $DC_\tau(T)$, and the length of the LCS is thus $r + \text{lcp}(T[p'_1..], T[p'_2..]) - 1$. In particular, this implies that

$$|\text{LCS}| \quad = \quad \max_{\substack{p'_1 \leq |T_1| \\ p'_2 > |T_1|+1}} \Big(\text{lcp}\big(RB(p'_1), RB(p'_2)\big) + \text{lcp}\big(T[p'_1..], T[p'_2..]\big) - 1 \Big) .$$

The -1 is necessary since a sampled suffix overlaps with the reversed block in one position. We will find the LCS by computing a pair of sampled positions $p^*_1 \leq |T_1|$, $p^*_2 > |T_1| + 1$ that maximises the above expression. Obviously, this can be done by performing two constant time range minimum queries for all $O((n/\sqrt{\tau})^2)$ pairs of sampled positions, but we want to do better.

The main idea of our algorithm is to exploit the observation that since $\text{lcp}(RB(p^*_1), RB(p^*_2)) \leq \tau$, it must hold that $\text{lcp}(T[p^*_1..], T[p^*_2..])$ is in the interval $[\ell_{max} - \tau + 1; \ell_{max}]$, where ℓ_{max} is the longest common prefix of two sampled suffixes of T_1 and T_2. Thus, we can ignore a lot of pairs with small lcp values.

First, we compute ℓ_{max} in $O(n/\sqrt{\tau})$ time by one scan of LCP_τ. We then compute the pair p^*_1, p^*_2 in τ rounds. In a round i, $0 \leq i \leq \tau - 1$, we only consider pairs $p'_1 \leq |T_1|, p'_2 > |T_1| + 1$ such that the length of the longest common prefix of $T[p'_1..]$ and $T[p'_2..]$ is at least $\ell = \ell_{max} - i$. Among these pairs we select the one maximising $\text{lcp}\big(RB(p'_1), RB(p'_2)\big)$.

The candidate pairs with a longest common prefix of length at least ℓ are located in disjoint intervals $I_1, I_2, \ldots, I_k$ of SA_τ. We compute these intervals by scanning LCP_τ to identify the maximal contiguous ranges with lcp values greater than or equal to ℓ. For each interval I_j we will find a pair $p'_1 \leq |T_1|, p'_2 > |T_1| + 1$ in I_j that maximises $\text{lcp}\big(RB(p'_1), RB(p'_2)\big)$. If $\text{lcp}\big(RB(p'_1), RB(p'_2)\big) + \ell - 1$ is greater than the maximum value seen so far, we store this value as the new maximum. See Figure 2 for an example.

Instead of searching the k intervals one by one, we process all intervals simultaneously. To do so, we first allocate an array A of size $n/\sqrt{\tau}$ and if r is the rank of a reversed block $RB(p)$, $p \in I_j$, we set $A[r]$ to be equal to j. We then scan A once and compute the longest common prefixes of every two consecutive reversed blocks ending at positions $p'_1 \leq |T_1|, p'_2 > |T_1| + 1$ from the same interval. We can do this if we for each interval I_j keep track of the rightmost r such that $A[r] = j$.

The intervals considered in each round are disjoint so each round takes $O(n/\sqrt{\tau})$ time and never uses more than $O(n/\sqrt{\tau})$ space. The total time is $O(n\sqrt{\tau})$ in addition to the $O(n\sqrt{\tau} + (n/\sqrt{\tau})\log(n/\sqrt{\tau}))$ time for the construction. Hence we have showed the following lemma:

Lemma 3. *Let $1 \leq \tau \leq n$. If the length of the longest common substring of T_1 and T_2 is at least τ, it can be computed in $O(n/\sqrt{\tau})$ space and $O(n\sqrt{\tau} + (n/\sqrt{\tau})\log n)$ time, where n is the total length of T_1 and T_2.*

3.2 A Solution for Short LCS

In the following we require that $\tau \leq n^{2/3}$, or, equivalently, that $\tau \leq n/\sqrt{\tau}$. Let us assume, for simplicity, that $n_1 = |T_1|$ is a multiple of τ. Note that if $|\mathrm{LCS}| \leq \tau$ then the LCS is a substring of one of the following strings: $T_1[1..2\tau], T_1[\tau + 1..3\tau], \ldots, T_1[n_1 - 2\tau + 1..n_1]$. Therefore, we can reduce the problem of computing the LCS to the problem of computing the longest substring of T_2 which occurs in at least one of these strings.

We divide the set $\mathcal{S} = \{T_1[1..2\tau], T_1[\tau + 1..3\tau], \ldots, T_1[n_1 - 2\tau + 1..n_1]\}$ into disjoint subsets $\mathcal{S}_i$, $i = 1, \ldots, \sqrt{\tau}$, such that the total length of strings in $\mathcal{S}_i$ is no more $2n/\sqrt{\tau}$ (note that we can do this since $\tau \leq n/\sqrt{\tau}$). For each $\mathcal{S}_i$ we compute the longest substring t_i^* of T_2 which occurs in one of the strings in S_i, and take the one of the maximal length.

To compute t_i^* for S_i we build the generalised suffix tree $ST(\mathcal{S}_i)$ for the strings in $\mathcal{S}_i$. We traverse $ST(\mathcal{S}_i)$ with T_2 as described in Lemma 1. Any common substring of T_2 and one of the strings in S_i will be a prefix of the label of some visited node in $ST(\mathcal{S}_i)$. It follows that t_i^* is the label of the node of maximal string depth visited during the traversal.

We now analyse the time and space complexity of the algorithm. Since the total length of the strings in $\mathcal{S}_i$ is at most $2n/\sqrt{\tau}$, the suffix tree can be built in $O(n/\sqrt{\tau})$ space and time. The traversal takes $O(n)$ time (see Lemma 1). Consequently, t_i^* can be found in $O(n/\sqrt{\tau})$ space and $O(n)$ time. By repeating for all $i = 1, \ldots, \sqrt{\tau}$, we obtain the following lemma:

Lemma 4. *Let $1 \leq \tau \leq n^{2/3}$. If the length of longest common substring of T_1 and T_2 is at most τ, it can be computed in $O(n/\sqrt{\tau})$ space and $O(n\sqrt{\tau})$ time, where n is the total length of T_1 and T_2.*

Combining the Solutions. By combining Lemma 3 and Lemma 4, we see that the LCS can be computed in $O\big(n/\sqrt{\tau}\big)$ space and $O\big(n\sqrt{\tau} + (n/\sqrt{\tau})\log n\big)$ time

for $1 \leq \tau \leq n^{2/3}$. Substituting $\tau = n^{2\varepsilon}$ the space bound becomes $O(n^{1-\varepsilon})$ and the time $O(n^{1+\varepsilon} + n^{1-\varepsilon} \log n)$, which is $O(n^{1+\varepsilon})$ for $\varepsilon > 0$. This concludes the proof of Theorem 1(i).

4 Longest Common Substring of Multiple Strings

In this section we prove Theorem 1(ii). Similar to the case of two strings, the algorithm consists of two procedures that both use space $O(n/\sqrt{\tau})$. The first one correctly computes the LCS if its length is at least $\tau' = \frac{1}{11}\tau \log^2 n$, while the second works if the length of the LCS is at most τ'. We then combine the solutions to obtain the desired trade-off. The choice of the specific separation value τ' comes from the fact that we need $\tau' \leq n$, and since the general solution for long LCS requires a data structure with a superlinear space bound.

4.1 A General Solution for Long LCS

Note that we cannot use the same idea that we use in the case of two strings since the property of difference cover samples (Lemma 2) does not necessarily hold for d positions. Instead we propose a different approach described below.

If $d > n/\sqrt{\tau}$, the algorithm returns an empty string and stops. This can be justified by the following simple observation.

Lemma 5. *If $d > n/\sqrt{\tau}$ then $|LCS| < \tau$.*

Proof. From $d > n/\sqrt{\tau}$ it follows that among any d strings from $T_1, T_2, \ldots, T_m$ there is at least one string shorter than $\sqrt{\tau}$. Therefore, the length of LCS is smaller than $\sqrt{\tau} < \tau$. □

This leaves us with the case where $d \leq n/\sqrt{\tau}$. We first construct the difference cover sample with parameter τ' for the string $T = T_1\$_1T_2\$_2 \cdots T_m\$_m$, where $\$_i$, $1 \leq i \leq m$, are special characters that do not occur in $T_1, T_2, \ldots, T_m$. We also construct the arrays and the range minimum query data structures described in Section 2.2 for computing longest common prefixes between pairs of sampled suffixes or pairs of reversed blocks in constant time.

Suppose that the LCS is a prefix of $T_i[p_i..]$, for some $1 \leq i \leq m$, $1 \leq p_i \leq |T_i|$. Then to compute |LCS| it is enough to find $(d-1)$ suffixes of distinct strings from $T_1, T_2, \ldots, T_m$ such that the lcp values for them and $T_i[p_i..]$ are maximal. The length of the LCS will be equal to the minimum of the lcp values. Below we show how to compute the minimum.

Let N_1 stand for zero, and N_i, $i \geq 2$, stand for the length of $T_1\$_1 \cdots T_{i-1}\$_{i-1}$. Consider the sampled positions $p_i^1, p_i^2, \ldots, p_i^z$ in an interval $[N_i + p_i, N_i + p_i + \tau']$ (see Figure 3).

From the property of the difference cover samples it follows that there is an integer $r < \tau'$ such that both $p'_i = (N_i + p_i) + r$ and $p'_j = (N_j + p_j) + r$ are in $DC_{\tau'}(T)$ — in particular, $p'_i = p_i^k$ for some k. Moreover, if $\mathrm{lcp}\big(T_i[p_i..], T_j[p_j..]\big) \geq$

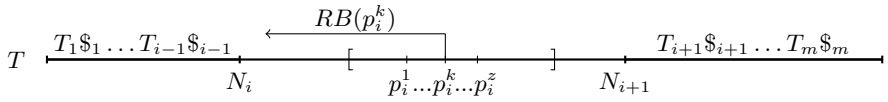

Fig. 3. Sampled positions $p_i^1, p_i^2, \ldots, p_i^z$ of T in an interval $[N_i + p_i, N_i + p_i + \tau']$, and a reversed block $RB(p_i^k)$.

τ', then the length of the longest common prefix of $RB(p_i^k)$ and $RB(p'_j)$ is at least $r = (p_i^k - N_i) - p_i$.

Let lcp_j^k be the maximum length of the longest common prefix of $T_i[p_i^k - N_i..]$ and $T_j[p'_j - N_j..]$, taken over all possible choices of p'_j, $N_j < p'_j \leq N_{j+1}$, such that $\mathrm{lcp}\big(RB(p_i^k), RB(p'_j)\big) \geq ((p_i^k - N_i) - p_i)$. For each k we define a list L^k to contain values $((p_i^k - N_i) - p_i) + \mathrm{lcp}_j^k - 1$, $j \neq i$, in decreasing order. Note that since the number of the sampled positions in $[N_i + p_i, N_i + p_i + \tau']$ is at most $\sqrt{1.5\tau'} + 6$ (see Section 2.2), the number of the lists does not exceed $\sqrt{1.5\tau'} + 6$ as well.

We first explain how we use the lists to obtain the answer and then how their elements are retrieved. The lists L^k are merged into a sorted list L until it contains values corresponding to suffixes of $(d-1)$ distinct strings from $T_1, T_2, \ldots, T_m$. The algorithm maintains a heap H_{val} on the values stored in the heads of the lists and a heap H_{id} on the distinct identifiers of strings already added to L. At each step it takes the maximum value in H_{val} and moves it from its list to L. Then it updates H_{val} and H_{id} and proceeds. The last value added to L will be equal to the length of the LCS.

We now explain how to retrieve values from L^k. Consider a set S of $|DC_{\tau'}(T)|$ coloured points in the plane, where a point corresponding to a position $p \in DC_{\tau'}(T)$ will have x-coordinate equal to the rank of $T[p..]$ in the lexicographic ordering of the sampled suffixes, y-coordinate equal to the rank of $RB(p)$ in the lexicographic ordering of the reversed blocks, and colour equal to the number of the string $T[p..]$ starts within.

We will show that after having retrieved the first $\ell - 1$ elements from L^k, the next element can be retrieved using $O(\log n)$ *coloured orthogonal range reporting* queries on the set S. For an integer ℓ and an axis-parallel rectangle $[a_1, b_1] \times [a_2, b_2]$, such a query reports ℓ points of distinct colours lying in the rectangle. We need only to consider the positions p such that $\mathrm{lcp}\big(RB(p_i^k), RB(p)\big) \geq ((p_i^k - N_i) - p_i)$. These positions form an interval I^k of the reversed block array, SA_τ^R. For each L^k we maintain a rectangle $R = [x_1; x_2] \times I^k$ such that $x_1 \leq x \leq x_2$, where x is the x-coordinate of the point corresponding to the position p_i^k. After the first $(\ell - 1)$ elements of L^k have been retrieved, R contains points of $(\ell - 1)$ colours besides i and $L^k[\ell - 1] = ((p_i^k - N_i) - p_i) + \mathrm{lcp}\big(x_1, x_2\big) - 1$, where $\mathrm{lcp}\big(x_1, x_2\big)$ is the longest common prefix of suffixes of T with ranks x_1 and x_2 (see Figure 4). To retrieve the next element we extend R until it contains points of ℓ colours not equal to i. We do this by extending either its left or right side until it includes a point of a new colour. We keep the rectangle that maximises

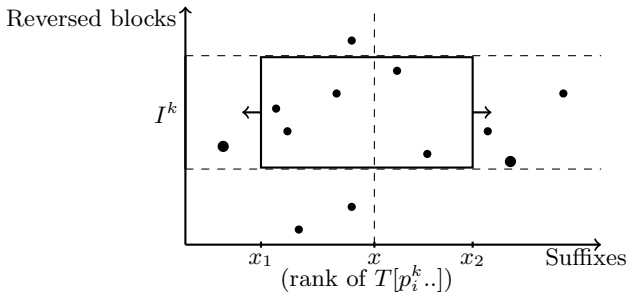

Fig. 4. Retrieving the ℓ^{th} element of L^k. A rectangle $R = [x_1; x_2] \times I^k$ contains points of $(\ell - 1)$ colours besides i. The two points of new colours shown in bold are the closest points of new colours from the left and from the right. We extend either the left or right side of the rectangle until it includes one of these points.

$\text{lcp}(x_1, x_2)$. Finding the two candidate rectangles can be done by performing two separate binary searches for the right and left sides using $O(\log n)$ coloured orthogonal range queries. Note that in each query at most ℓ points are to be reported.

The procedure described above is repeated for all $1 \le i \le m$ and $1 \le p_i \le |T_i|$. The maximum of the retrieved values will be equal to the length of the LCS. We can compute the LCS itself, too, if we remember i and p_i on which the maximum is achieved.

Lemma 6. *Let $1 \le \tau \le 11n/\log^2 n$, and let LCS denote the longest substring that appears in at least d of the strings $T_1, T_2, \ldots, T_m$ of total length n. In the case where $|LCS| \ge \frac{1}{11}\tau \log^2 n$, the LCS can be found in $O(n/\sqrt{\tau})$ space and $O(nd\sqrt{\tau}\log^2 n(\log^2 n + d))$ time.*

Proof. If $d > n/\sqrt{\tau}$, the algorithm returns an empty string and thus is correct. Otherwise, $\tau' = \frac{1}{11}\tau \log^2 n \le n$, and correctness of the algorithm follows from its description. The data structures for performing constant time lcp computations require $O(n/\sqrt{\tau})$ space and can be built in $O(n\sqrt{\tau}\log n)$ time.

Suppose that i and p_i are fixed. Each interval I^k can be found using $O(\log n)$ lcp computations. To perform coloured orthogonal range queries on the set S of size $|DC_{\tau'}(T)| = O(n/(\sqrt{\tau}\log n))$, we use the data structure [8] that can be constructed in $O(|S|\log^2|S|) = O((n\log n)/\sqrt{\tau})$ time and $O(|S|\log|S|) = O(n/\sqrt{\tau})$ space and allows to report ℓ points of distinct colours in time $O(\log^2|S| + \ell) = O(\log^2 n + \ell)$. Thus retrieving $L^k[\ell]$ takes time $O(\log n(\log^2 n + \ell))$. The merge stops after retrieving at most d elements from each of the $O(\sqrt{\tau'})$ lists, which will take $O(d\sqrt{\tau'}\log n(\log^2 n + d)) = O(d\sqrt{\tau}\log^2 n(\log^2 n + d))$ time.

Merging the lists into L will take $O(\log\tau' + \log d)$ time per element, i.e., $O(d\sqrt{\tau'}(\log\tau' + \log d)) = O(d\sqrt{\tau}\log^{3/2} n)$ time in total, and $O(\sqrt{\tau'} + d) = O(n/\sqrt{\tau})$ space (remember that we are in the case $d \le n/\sqrt{\tau}$). Therefore, computing the longest prefix of $T_i[p_i..]$ which occurs in at least $(d-1)$ other strings will take $O(d\sqrt{\tau}\log^2 n(\log^2 n + d))$ time. The lemma follows. □

4.2 A General Solution for Short LCS

We start by proving the following lemma:

Lemma 7. *Given input strings $T_1, T_2, \ldots, T_m$ of total length n and a string S of length $|S|$. The longest substring t of S that appears in at least d of the input strings can be found in $O((|S| + n) \log |t|)$ time and $O(|S|)$ space.*

Proof. We prove that there is an algorithm that takes an integer i, and in $O(|S|+n)$ time and $O(|S|)$ space either finds an i-length substring of S that occurs in at least d input strings, or reports that no such substring exists. The lemma then follows, since by running the algorithm $O(\log |t|)$ times we can do an exponential search for the maximum value of i.

We construct the algorithm as follows. First we build the suffix tree $ST(S)$ for the string S, together with all suffix links. For every node of the suffix tree we store a pointer to its ancestor of string depth i (all such pointers can be computed in $O(|S|)$ time by post-processing the tree). Besides, for every node $v \in ST(S)$ of string depth i (explicit or implicit), we store a counter $c(v)$ and an integer $id(v)$, both initially set to zero. These nodes correspond exactly to the i-length substrings of S, and we will use $c(v)$ to count the number of distinct input strings that the label of v occurs in. To do this, we traverse $ST(S)$ with the input strings $T_1, T_2, \ldots, T_m$ one at a time as described in Lemma 1. When matching a character a of T_j, we always check if a node v of string depth i above our current location has $id(v) < j$. In that case, we increment the counter $c(v)$ and set $id(v) = j$ to ensure that the counter is only incremented once for T_j.

To prove the correctness note that for any i-length substring ℓ of T_j that also occurs in S there exists a node of $ST(T)$ labelled by it, and one of the descendants of this node will be visited during the matching process of T_j (see Lemma 1). The converse is also true, because any node $v' \in ST(T)$ visited during the traversal implies that all prefixes of the label of v' occur in T_j.

The suffix tree for S can be constructed in $O(|S|)$ time and space. The traversal with T_j can be implemented to take time $O(|T_j|)$, i.e., $O(n)$ time for all the input strings. In addition to the suffix tree, at most $|S|$ constant space counters are stored. Thus the algorithm requires $O(n + |S|)$ time and $O(|S|)$ space. □

We now describe the algorithm for finding the LCS when $|\text{LCS}| \leq \tau' = \frac{1}{11}\tau \log^2 n$. Consider the partition of T into substrings of length $\delta n/\sqrt{\tau}$ overlapping in τ' positions, where δ is a suitable constant. Assuming that $\tau \leq n^{2/3-\gamma}$ for some constant $\gamma > 0$, implies that these strings will have length at least $2\tau'$, and thus the LCS will be a substring of one of them. We examine the strings one by one and apply Lemma 7 to find the longest substring that occurs in at least d input strings. It follows that we can check one string in $O(n/\sqrt{\tau})$ space and $O(n \log n)$ time, so by repeating for all $O(\sqrt{\tau})$ strings, we have:

Lemma 8. *Let $1 \leq \tau \leq n^{2/3-\gamma}$ for some constant $\gamma > 0$, and let LCS denote the longest substring that appear in at least d of the strings $T_1, T_2, \ldots, T_m$ of total length n. If $|LCS| \leq \frac{1}{11}\tau \log^2 n$, the LCS can be found in $O(n/\sqrt{\tau})$ space and $O(\sqrt{\tau} n \log n)$ time.*

Combining the Solutions. Our specific choice of separation value ensures that the assumption on τ of Lemma 8 implies the assumption of Lemma 6 (because $n^{2/3-\gamma} \leq 11n/\log^2 n$ for all n and $\gamma > 0$). Thus by combining the two solutions the LCS can be computed in $O(n/\sqrt{\tau})$ space and $O(d\sqrt{\tau}n \log^2 n(\log^2 n + d))$ time for $1 \leq \tau \leq n^{2/3-\gamma}$, $\gamma > 0$. Substituting $\tau = n^{2\varepsilon}$, we obtain the bound stated by Theorem 1(ii) with the requirement that $0 \leq \varepsilon < 1/3$.

5 Open Problems

We conclude with some open problems. Is it possible to extend the trade-off range of our solutions to ideally $0 \leq \varepsilon \leq 1/2$? Can the time bound for the general LCS problem be improved so it fully generalises the solution for two strings? The difference cover technique requires $\Omega(\sqrt{n})$ space, so the most interesting question is perhaps whether the LCS problem can be solved deterministically in $O(n^{1-\varepsilon})$ space and $O(n^{1+\varepsilon})$ time for any $0 \leq \varepsilon \leq 1$?

Acknowledgements. T. Starikovskaya has been partly supported by a grant 10-01-93109-CNRS-a of the Russian Foundation for Basic Research and by Dynasty foundation.

References

1. M.A. Bender and M. Farach-Colton. The LCA Problem Revisited. In *Proc. 4th LATIN (LNCS 1776)*, pages 88–94, 2000.
2. P. Bille, I.L. Gørtz, B. Sach, and H.W. Vildhøj. Time-Space Trade-Offs for Longest Common Extensions. In *Proc. 23rd CPM (LNCS 7354)*, pages 293–305, 2012.
3. S. Burkhardt and J. Kärkkäinen. Fast Lightweight Suffix Array Construction and Checking. In *Proc. 14th CPM (LNCS 2676)*, pages 55–69, 2003.
4. C.J. Colbourn and A.C.H. Ling. Quorums from difference covers. *Inf. Process. Lett.*, 75(1-2):9–12, 2000.
5. H.N. Gabow, J.L. Bentley, and R.E. Tarjan. Scaling and Related Techniques for Geometry Problems. In *Proc. 16th STOC*, pages 135–143, 1984.
6. D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology.* Cambridge University Press, New York, NY, USA, 1997.
7. L. Ilie, G. Navarro, and L. Tinta. The longest common extension problem revisited and applications to approximate string searching. *J. Discrete Algorithms*, 8(4):418–428, 2010.
8. R. Janardan and M. Lopez. Generalized Intersection Searching Problems. *Int. J. Comput. Geom. Appl.*, 3(1):39–69, 1993.
9. R.M. Karp and M.O. Rabin. Efficient Randomized Pattern-Matching Algorithms. *IBM J. Res. Dev.*, 31(2):249–260, 1987.
10. D.E. Knuth, J.H. Morris, and V.R. Pratt. Fast Pattern Matching in Strings. *SIAM J. Comput.*, 6(2):323–350, 1977.
11. S.J. Puglisi and A. Turpin. Space-Time Tradeoffs for Longest-Common-Prefix Array Computation. In *Proc. 19th ISAAC (LNCS 5369)*, pages 124–135, 2008.