Longest Common Substring Made Fully Dynamic

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

Panagiotis Charalampopoulos
Department of Informatics, King's College London, London, UK
Efi Arazi School of Computer Science, The Interdisciplinary Center Herzliya, Herzliya, Israel
panagiotis.charalampopoulos@kcl.ac.uk

Solon P. Pissis
CWI, Amsterdam, The Netherlands
solon.pissis@cwi.nl

Jakub Radoszewski
Institute of Informatics, University of Warsaw, Warsaw, Poland
Samsung R&D Institute, Warsaw, Poland
jrad@mimuw.edu.pl

─── **Abstract** ───

Given two strings S and T, each of length at most n, the longest common substring (LCS) problem is to find a longest substring common to S and T. This is a classical problem in computer science with an $\mathcal{O}(n)$-time solution. In the fully dynamic setting, edit operations are allowed in either of the two strings, and the problem is to find an LCS after each edit. We present the first solution to this problem requiring sublinear time in n per edit operation. In particular, we show how to find an LCS after each edit operation in $\tilde{\mathcal{O}}(n^{2/3})$ time, after $\tilde{\mathcal{O}}(n)$-time and space preprocessing. [1]

This line of research has been recently initiated in a somewhat restricted dynamic variant by Amir et al. [SPIRE 2017]. More specifically, they presented an $\tilde{\mathcal{O}}(n)$-sized data structure that returns an LCS of the two strings after a single edit operation (that is reverted afterwards) in $\tilde{\mathcal{O}}(1)$ time. At CPM 2018, three papers (Abedin et al., Funakoshi et al., and Urabe et al.) studied analogously restricted dynamic variants of problems on strings. We show that the techniques we develop can be applied to obtain fully dynamic algorithms for all of these variants. The only previously known sublinear-time dynamic algorithms for problems on strings were for maintaining a dynamic collection of strings for comparison queries and for pattern matching, with the most recent advances made by Gawrychowski et al. [SODA 2018] and by Clifford et al. [STACS 2018].

As an intermediate problem we consider computing the solution for a string with a given set of k edits, which leads us, in particular, to answering *internal* queries on a string. The input to such a query is specified by a substring (or substrings) of a given string. Data structures for answering internal string queries that were proposed by Kociumaka et al. [SODA 2015] and by Gagie et al. [CCCG 2013] are used, along with new ones, based on ingredients such as the suffix tree, heavy-path decomposition, orthogonal range queries, difference covers, and string periodicity.

─────────

[1] The $\tilde{\mathcal{O}}(\cdot)$ notation suppresses $\log^{\mathcal{O}(1)} n$ factors.

1 Introduction

Given two strings S and T, each of length at most n, the longest common substring (LCS) problem, also known as the longest common factor problem, is to find a longest substring common to S and T. This is a classical problem in theoretical computer science. Knuth had conjectured that the LCS problem was in $\Omega(n \log n)$. In 1973 Weiner solved it in the optimal $\mathcal{O}(n)$ time [49] designing a data structure that was later called the suffix tree (see also [21]). Knuth declared Weiner's algorithm the "Algorithm of the Year" [11]. Since $\mathcal{O}(n)$ time is optimal for this problem, a series of studies have been dedicated in improving the working space [37, 44]. The LCS problem has also been studied under Hamming and edit distance. We refer the interested reader to [1, 43, 47, 16, 46, 12] and references therein.

In [43], Starikovskaya mentions that an answer to the LCS problem "is not robust and can vary significantly when the input strings are changed even by one character", implicitly posing the following question: *Can we compute an LCS after editing S or T in $o(n)$ time?*

▶ **Example 1.** The length of an LCS of S and T below is *doubled* when substitution $S[4] :=$ a is performed. The next substitution, $T[3] :=$ b, *halves* the length of an LCS.

$$S = \text{c}\underline{\text{aab}}\text{aaa} \qquad S[4] := \text{a} \qquad S = \text{c}\underline{\text{aaaaaa}} \qquad T[3] := \text{b} \qquad S = \text{c}\underline{\text{aaaaaa}}$$

$$T = \text{aaaa}\underline{\text{aab}} \qquad\qquad\qquad T = \underline{\text{aaaaaa}}\text{b} \qquad\qquad\qquad T = \text{aab}\underline{\text{aaa}}\text{b}$$

This question poses the challenge of dynamically updating the suffix tree in the presence of edit operations (i.e. insertions, deletions and substitutions), which remains the main obstacle for answering this type of questions.

Amir et al. [8] introduced a restricted dynamic variant, where any *single* edit operation is allowed and is reverted afterwards. We call this problem LCS AFTER ONE EDIT. Amir et al. presented an $\tilde{\mathcal{O}}(n)$-sized data structure that can be constructed in $\tilde{\mathcal{O}}(n)$ time supporting $\tilde{\mathcal{O}}(1)$-time computation of an LCS, after one edit operation is applied on S. This work initiated a new line of research on analogously restricted dynamic variants of problems on strings [26, 27, 48]. Abedin et al. [3] improved the complexities of the data structure proposed by Amir et al. [8] by $\log^{\mathcal{O}(1)} n$ factors. Two other restricted variants of the dynamic LCS problem were considered by Amir and Boneh in [5]. In both variants substitutions were allowed in one of the strings; one was of decremental nature and in the other one the complexity was parameterized by the period of the static string.

In this paper we make substantial progress: we show a strongly sublinear-time solution for the general version of the problem, namely, the fully dynamic case of the LCS problem. Given two strings S and T, the problem is to answer the following type of queries in an on-line manner: perform an edit operation (substitution, insertion, or deletion) on S or on T and then return an LCS of the new S and T. We call this problem FULLY DYNAMIC LCS. Below we mention some of the known results on dynamic problems on strings.

Dynamic Pattern Matching. Finding all *occ* occurrences of a pattern of length m in a *static* text can be performed in the optimal $\mathcal{O}(m + occ)$ time using suffix trees, which can be constructed in linear time [49, 21]. In the fully dynamic setting, the problem is to compute the new set of occurrences when allowing for edit operations anywhere on the text. A considerable amount of work has been carried out on this problem [31, 22, 23]. The first data structure with polylogarithmic update time and time-optimal queries was shown by Sahinalp and Vishkin [41]. The update time was later improved by Alstrup et al. [4] at the expense of slightly suboptimal query time. The state of the art is the data structure by

Gawrychowski et al. [29] supporting time-optimal queries with $\mathcal{O}(\log^2 n)$ time for updates. Clifford et al. [18] have recently shown upper and lower bounds for variants of exact matching with wildcard characters, inner product, and Hamming distance.

Dynamic String Collection with Comparison.   The problem is to maintain a dynamic collection $\mathcal{W}$ of strings of total length n supporting the following operations: adding a string to $\mathcal{W}$, adding the concatenation of two strings from $\mathcal{W}$ to $\mathcal{W}$, splitting a string from $\mathcal{W}$ and adding the two residual strings in $\mathcal{W}$, and returning the length of the longest common prefix of two strings from $\mathcal{W}$. This line of research was initiated by Sundar and Tarjan [45]. Data structures supporting updates in polylogarithmic time were presented by Mehlhorn et al. [40] and Alstrup et al. [4]. Finally, Gawrychowski et al. [30] proposed an optimal solution.

Longest Palindrome and Longest Lyndon Substring.   A string is called *palindrome* if it the same as its reverse. A string is called *Lyndon* if it is smaller lexicographically than all its suffixes [38]. Computing a longest palindrome and a longest Lyndon substring of a string after a single edit have been recently studied in [26] (see also [27]) and in [48], respectively.

Maintaining Repetitions.   *Squares* are strings of the form XX. In [7], the authors show how to maintain squares in a dynamic string S of length n in $n^{o(1)}$ time per operation. A modification of this algorithm, with the same time complexity per operation, allows them to determine in $\tilde{\mathcal{O}}(1)$ time whether a queried substring of S is periodic, and if so, compute its period.

Our Results.   We give the first fully dynamic algorithm for the LCS problem that works in *strongly sublinear* time per edit operation in any of the two strings. Specifically, for two strings, each of length up to n, it computes an LCS after each edit operation in $\tilde{\mathcal{O}}(n^{2/3})$ time after $\tilde{\mathcal{O}}(n)$-time and space preprocessing. To ease the comprehension of the algorithm for FULLY DYNAMIC LCS, we first show a solution of an auxiliary problem called LCS AFTER ONE SUBSTITUTION PER STRING, where a single substitution is allowed in each of the strings and is reverted afterwards, with $\tilde{\mathcal{O}}(1)$-time queries after $\tilde{\mathcal{O}}(n)$-time and space preprocessing.

Notably, we showcase the applicability of our techniques to other string problems in the fully dynamic setting. We present a fully dynamic algorithm for computing a longest repeat of a string S of length n, i.e. a longest substring occurring more than once in S, in $\tilde{\mathcal{O}}(n^{2/3})$ time. We also present a fully dynamic algorithm for computing a longest palindrome substring of a string S requiring $\tilde{\mathcal{O}}(\sqrt{n})$ time per edit. Finally, we present a fully dynamic algorithm, requiring $\tilde{\mathcal{O}}(\sqrt{n})$ time per edit, for computing a longest Lyndon substring of string S as well as maintaining a representation of the Lyndon factorization of S that allows us to efficiently extract the t-th element of the factorization in $\tilde{\mathcal{O}}(1)$ time.

Our data structure is randomized due to the use of data structures for dynamic strings [30] and internal pattern matching [35]; the latter can be derandomized [34].

Roadmap.   Section 2 provides the necessary definitions and notation used throughout as well as the standard algorithmic toolbox for string processing and the general scheme of our approach. In Section 3 we show an optimal, up to polylogarithmic factors, solution for LCS AFTER ONE SUBSTITUTION PER STRING. In Section 4 we show our main result: a solution for FULLY DYNAMIC LCS. Some technical details, including details on several special cases of internal LCS queries, are omitted in this version. A brief overview of our fully dynamic algorithms for computing the longest repeat, the longest palindrome, and the longest Lyndon substring of a string is provided in Section 5. We conclude this work in Section 6.

2   Preliminaries

Strings.   Let $S = S[1]S[2]\ldots S[n]$ be a *string* of length $|S| = n$ over an integer alphabet $\Sigma = \{1,\ldots,n^{\mathcal{O}(1)}\}$. The elements of Σ are called *characters*. For two positions i and j on S, we denote by $S[i\mathinner{\ldotp\ldotp}j] = S[i]\ldots S[j]$ the substring of S that starts at position i and ends at position j (it is empty if $i > j$). A substring of S is represented in $\mathcal{O}(1)$ space by specifying the indices i and j. A prefix $S[1\mathinner{\ldotp\ldotp}j]$ is denoted by $S^{(j)}$ and a suffix $S[i\mathinner{\ldotp\ldotp}n]$ is denoted by $S_{(i)}$. A substring of S is called *proper* if it is shorter than S. We denote the *reverse string* of S by $S^R = S[n]S[n-1]\ldots S[1]$. By ST, S^k, and S^∞ we denote the concatenation of strings S and T, k copies of string S, and infinitely many copies of string S, respectively. If a string B is both a proper prefix and a proper suffix of string S, then B is called a *border* of S. A positive integer p is called a *period* of S if $S[i] = S[i+p]$ for all $i = 1,\ldots,n-p$. String S has a period p if and only if it has a border of length $n-p$. We refer to the smallest period as *the period* of the string and, analogously, to the longest border as *the border* of the string.

The *suffix tree* $\mathcal{T}(S)$ of string S is a compact trie representing all suffixes of S. The suffix tree of a string of length n over an integer alphabet can be constructed in $\mathcal{O}(n)$ time and space [21]. By $\mathsf{lcpstring}(S,T)$ we denote the longest common prefix of S and T, by $\mathsf{lcp}(S,T)$ we denote $|\mathsf{lcpstring}(S,T)|$, and by $\mathsf{lcp}(r,s)$ we denote $\mathsf{lcp}(S_{(r)}, S_{(s)})$. Further by $\mathsf{lcsstring}(S,T)$ we denote the longest common suffix of S and T. An $\mathcal{O}(n)$-sized lowest common ancestor data structure can be constructed over the suffix tree of S in $\mathcal{O}(n)$ time [14], supporting $\mathsf{lcp}(r,s)$-queries in $\mathcal{O}(1)$ time. A symmetric construction on S^R (the reverse of S) can answer the so-called *longest common suffix* (lcs) queries in the same complexity. The lcp and lcs queries are also known as *longest common extension* (LCE) queries.

General Scheme and Relation to Internal Pattern Matching.   The scheme of our approach for most of the considered dynamic problems on strings is as follows. Let the input be a string S of length n (in the case of the LCS problem, this can be the concatenation of the input strings S and T separated by a delimiter). We construct a data structure that answers the following type of queries: given k edit operations on S, compute the answer to a particular problem on the resulting string S'. Assuming that the data structure occupies $\mathcal{O}(s_n)$ space, answers queries for k edits in time $\mathcal{O}(q_n(k))$ and can be constructed in time $\mathcal{O}(t_n)$ ($s_n \geq n$ and $q_n(k) \geq k$ is non-decreasing with respect to k), this data structure can be used to design a dynamic algorithm that preprocesses the input string in time $\mathcal{O}(t_n)$ and answers queries dynamically under edit operations in amortized time $\mathcal{O}(q_n(\kappa))$, where κ is such that $q_n(\kappa) = (t_n + n)/\kappa$, using $\mathcal{O}(s_n)$ space. The query time can be made worst-case using *time slicing*: for $s_n, t_n = \tilde{\mathcal{O}}(n)$ and $q_n(k) = \tilde{\mathcal{O}}(k)$ we obtain a fully dynamic algorithm with $\tilde{\mathcal{O}}(\sqrt{n})$-time queries, whereas for $q_n(k) = \tilde{\mathcal{O}}(k^2)$ the query time is $\tilde{\mathcal{O}}(n^{2/3})$.

A *k-substring* of a string S is a concatenation of k strings, each of which is either a substring of S or a single character. A k-substring of S can be represented in $\mathcal{O}(k)$ additional space using a doubly-linked list if the string S itself is stored. The string S after k subsequent edit operations can be represented as a $(2k+1)$-substring due to the following lemma.

▶ **Lemma 2.**   *Let S' be a k-substring of S and S'' be S' after a single edit operation. Then S'' is a $(k+2)$-substring of S. Moreover, S'' can be computed from S' in $\mathcal{O}(k)$ time.*

Proof.   Let $S' = F_1 \ldots F_k$ where each F_i is either a substring of S or a single character. We traverse the list of substrings until we find the substring F_i such that the edit operation takes place at the j-th character of F_i. As a result, F_i is decomposed into a prefix and a suffix, potentially with a single character inserted in between in case of insertion or substitution. The resulting string S'' is a $(k+2)$-substring of S.   ◀

Thus the fully dynamic version reduces to designing a data structure over a string S of length n that computes the result of a specific problem on a k-substring $F_1 \ldots F_k$ of S. For the considered problems we aim at computing the longest substring of S that satisfies a certain property. Then there are two cases. Case 1: the sought substring occurs inside one of the substrings F_i (or each of its two occurrences satisfies this property in case of the LCS and the longest repeat problems). Case 2: it contains the boundary between some two substrings F_i and F_{i+1}. Case 1 requires to compute the solution to a certain problem on a substring or substrings of a specified string. This is the so-called *internal* model of queries; this name was coined by Kociumaka et al. in [35]. We call Case 2 *cross-substring* queries. Due to string periodicity, certain internal queries arise in cross-substring queries as well.

3 LCS After One Substitution Per String

Let us now consider an extended version of the LCS AFTER ONE EDIT problem, for simplicity restricted to substitutions.

LCS AFTER ONE SUBSTITUTION PER STRING
Input: Two strings S and T of length at most n
Query: For given indices i, j and characters α and β, compute $\mathsf{LCS}(S', T')$ where S' is S after substitution $S[i] := \alpha$ and T' is T after substitution $T[j] := \beta$

To solve this problem we consider three cases depending on whether an occurrence of the LCS contains any of the changed positions in S and T. We prove the following result.

▶ **Theorem 3.** LCS AFTER ONE SUBSTITUTION PER STRING *can be computed in* $\tilde{\mathcal{O}}(1)$ *time after* $\tilde{\mathcal{O}}(n)$*-time and space preprocessing.*

3.1 LCS Contains No Changed Position

We use the following lemma for a special case of internal LCS queries. Its proof is deferred to the full version. In the fully dynamic algorithm a less restrictive approach is necessary.

▶ **Lemma 4.** *Let S and T be two strings of length at most n. After $\mathcal{O}(n \log^2 n)$-time and $\mathcal{O}(n \log n)$-space preprocessing, an LCS between any prefix or suffix of S and prefix or suffix of T can be computed in $\mathcal{O}(\log n)$ time.*

It suffices to apply internal LCS queries of Lemma 4 four times: each time for one of $S^{(i-1)}, S_{(i+1)}$ and one of $T^{(j-1)}, T_{(j+1)}$.

### 3.2 LCS Contains a Changed Position in Exactly One of the Strings

We use the following lemma that encapsulates one of the main techniques of [8]. It involves computing so-called *ranges* of substrings in the *generalized suffix array* of S and T and it relies on a result by Fischer et al. [24].

▶ **Lemma 5.** *Let S and T be strings of length at most n. After $\mathcal{O}(n \log \log n)$-time and $\mathcal{O}(n)$-space preprocessing, given two substrings P and Q of S or T, we can compute:*
(a) *a substring of T equal to PQ, if it exists, in $\mathcal{O}(\log \log n)$ time;*
(b) *the longest substring of T that is a prefix (or a suffix) of PQ in $\mathcal{O}(\log n \log \log n)$ time.*

We now show how to compute the longest substring that contains the position i in S, but not the position j in T (the opposite case is symmetric). We first use Lemma 5(b) to compute two substrings, U and V, of T in $\mathcal{O}(\log n \log \log n)$ time:
- U is the longest substring of T that is equal to a suffix of $S[1 \ldots i-1]$;
- V is the longest substring of T that is equal to a prefix of $\alpha S[i+1 \ldots |S|]$.
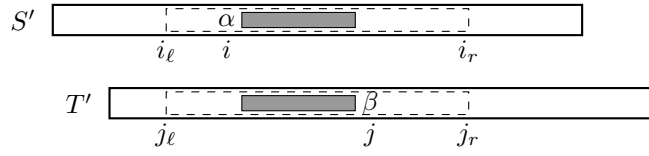
Figure 1 Occurrences of an LCS of S' and T' containing both changed positions are denoted by dashed rectangles. Occurrences of U at which an LCS is aligned are denoted by gray rectangles.

Our task then reduces to computing the longest substring of UV that crosses the boundary between U and V and is a substring of $T^{(j-1)}$ or of $T_{(j+1)}$. We can compute it using the following type of queries.

THREE SUBSTRINGS LCS
Input: A string T
Query: Given three substrings U, V, and W of T, compute the longest substring XY of W such that X is a suffix of U and Y is a prefix of V

Indeed, it suffices to ask two THREE SUBSTRINGS LCS queries: one with $W = T^{(j-1)}$ and one with $W = T_{(j+1)}$.

A solution to a special case of THREE SUBSTRINGS LCS queries with $W = T$ was already implicitly presented by Amir et al. in [8]. It is based on the *heaviest induced ancestors* (HIA) problem on trees, introduced by Gagie et al. [28], applied to the suffix tree of T. We generalize the HIA queries and use them to answer general THREE SUBSTRINGS LCS queries. The data structure for answering our generalization of HIA queries turns out to be one of the most technical parts of the paper. It relies on the construction of multidimensional grids for pairs of heavy paths (in heavy-path decompositions [42]) of the involved trees. Each query can be answered by interpreting the answer of $\mathcal{O}(\log^2 n)$ orthogonal range maximum queries over such grids.

▶ **Lemma 6.** *Let T be a string of length at most n. After $\tilde{\mathcal{O}}(n)$-time preprocessing, we can answer* THREE SUBSTRINGS LCS *queries in $\tilde{\mathcal{O}}(1)$ time.*

3.3 LCS Contains a Changed Position in Each of the Strings

A Prefix-Suffix Query gets as input two substrings X and Y of a string S of length n and an integer d and returns the lengths of all prefixes of X of length between d and $2d$ that are suffixes of Y. It is known that such a query returns an arithmetic sequence and if it has at least three elements, then its difference equals the period of all the corresponding prefixes-suffixes. Moreover, Kociumaka et al. [35] show that Prefix-Suffix Queries can be answered in $\mathcal{O}(1)$ time using a data structure of $\mathcal{O}(n)$ size, which can be constructed in $\mathcal{O}(n)$ time. By considering $X = Y = U$, this implies the two respective points of the lemma below.

▶ **Lemma 7.**
(a) *For a string U of length m, the set $\mathcal{B}_r(U)$ of border lengths of U between 2^r and $2^{r+1} - 1$ is an arithmetic sequence. If it has at least three elements, all the corresponding borders have the same period, equal to the difference of the sequence.*
(b) *[35] Let S be a string of length n. For any substring U of S and integer r, the arithmetic sequence $\mathcal{B}_r(U)$ can be computed in $\mathcal{O}(1)$ time after $\mathcal{O}(n)$-time and space preprocessing.*
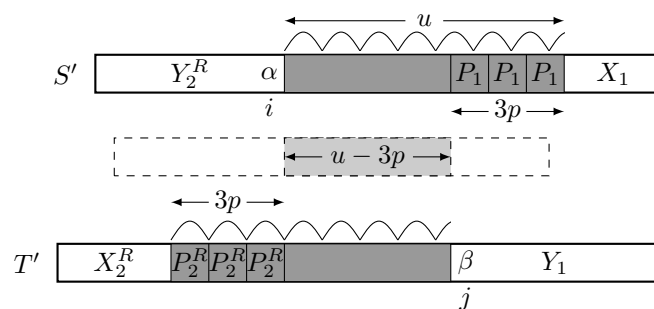
Figure 2 A border of length u is denoted by dark gray rectangles. An LCS aligned at a border of length $u - 3p$, which is in the same arithmetic sequence, is denoted by the dashed rectangle.

We next show an algorithm that finds a longest string $S'[i_\ell \mathinner{.\,.} i_r] = T'[j_\ell \mathinner{.\,.} j_r]$ such that $i_\ell \leq i \leq i_r$ and $j_\ell \leq j \leq j_r$ for the given indices i, j. Let us assume that $i - i_\ell \leq j - j_\ell$; the symmetric case can be treated analogously. We have that $U \stackrel{\text{def}}{=} S'[i+1 \mathinner{.\,.} i_\ell + j - j_\ell - 1] = T'[j_\ell + i - i_\ell + 1 \mathinner{.\,.} j - 1]$ as shown in Figure 1. ($U = \varepsilon$ can correspond to $i - i_\ell = j - j_\ell$ or $i - i_\ell + 1 = j - j_\ell$, so both these cases need to be checked.) Note that these substrings do not contain any changed position. Any such U is a prefix of $S_{(i+1)}$ and a suffix of $T^{(j-1)}$; let U_0 denote the longest such string. Then, the possible candidates for U are U_0 and all its borders. For a border U of U_0, we say that $\mathsf{lcsstring}(S'^{(i)}, T'^{(j-|U|-1)}) \, U \, \mathsf{lcpstring}(S'_{(i+|U|+1)}, T'_{(j)})$ is an *LCS aligned at U*. We compute U_0 in time $\mathcal{O}(\log n)$ by asking Prefix-Suffix Queries for $X = S_{(i+1)}$, $Y = T^{(j-1)}$ in $S\#T$ and $d = 2^r$ for all $r = 0, 1, \ldots, \lfloor \log j \rfloor$. We then consider the borders of U_0 in arithmetic sequences of their lengths; see Lemma 7. If an arithmetic sequence has at most two elements, we compute an LCS aligned at each of the borders in $\mathcal{O}(1)$ time by the above formula using LCE queries. Otherwise, let p be the difference of the arithmetic sequence, ℓ be its length, and u be its maximum element. Further let:

$$X_1 = S'_{(i+u+1)}, \qquad Y_1 = T'_{(j)}, \qquad P_1 = S'[i + u - p + 1 \mathinner{.\,.} i + u],$$
$$X_2^R = T'^{(j-u-1)}, \quad Y_2^R = S'^{(i)}, \quad P_2^R = T'[j - u \mathinner{.\,.} j - u + p - 1].$$

The setting is presented in Figure 2. It can be readily verified (inspect Figure 2) that a longest common substring aligned at the border of length $u - wp$, for $w \in [0, \ell - 1]$, is equal to

$$\mathsf{lcs}(X_2^R (P_2^R)^w, Y_2^R) + u - wp + \mathsf{lcp}(P_1^w X_1, Y_1) = \mathsf{lcp}(P_2^w X_2, Y_2) + \mathsf{lcp}(P_1^w X_1, Y_1) + u - wp$$

which we further denote by $g(w)$. Thus, a longest LCS aligned at a border whose length is in this arithmetic sequence is $\max_{w=0}^{\ell-1} g(w)$. The following observation facilitates efficient evaluation of this formula.

▶ **Observation 8.** *For any strings P, X, Y, the function $f(w) = \mathsf{lcp}(P^w X, Y)$ for integer $w \geq 0$ is piecewise linear with at most three pieces. Moreover, if P, X, Y are substrings of a string S, then the exact formula of f can be computed with $\mathcal{O}(1)$ LCE queries on S.*

Proof. Let $a = \mathsf{lcp}(P^\infty, X)$, $b = \mathsf{lcp}(P^\infty, Y)$, and $p = |P|$. Then:

$$f(w) = \begin{cases} a + wp & \text{if } a + wp < b \\ w + \mathsf{lcp}(X, Y[aw + 1 \mathinner{.\,.} |Y|]) & \text{if } a + wp = b \\ b & \text{if } a + wp > b. \end{cases}$$

Note that a can be computed from $\mathsf{lcp}(P, X)$ and $\mathsf{lcp}(X, X[p+1 \mathinner{.\,.} |X|])$, and b analogously. Thus if P, X, Y are substrings of S, five LCE queries on S suffice.   ◀

By Observation 8, $g(w)$ can be expressed as a piecewise linear function with $\mathcal{O}(1)$ pieces. Moreover, its exact formula can be computed using $\mathcal{O}(1)$ LCE queries on $S'\#T'$, hence, in $\mathcal{O}(1)$ time using LCE queries. This allows to compute $\max_{w=0}^{\ell-1} g(w)$ in $\mathcal{O}(1)$ time. Each arithmetic sequence is processed in $\mathcal{O}(1)$ time. The global maximum that contains both changed positions is the required answer. Thus the query time in this case is $\mathcal{O}(\log n)$ and the preprocessing requires $\mathcal{O}(n)$ time and space.

By combining the results of Sections 3.1 to 3.3, we arrive at Theorem 3.

4   Fully Dynamic LCS

In this section we assume that the sought LCS has length at least 2. The case that it is of unit or zero length can be easily treated separately. We use the following auxiliary problem that generalizes LCS AFTER ONE SUBSTITUTION PER STRING into the case of k edit operations:

(k_1, k_2)-SUBSTRING LCS
Input: Two strings S and T of length at most n
Query: Compute $\mathsf{LCS}(S', T')$ where $S' = F_1 \ldots F_{k_1}$ is a k_1-substring of S, $T' = G_1 \ldots G_{k_2}$ is a k_2-substring of T, and $k_1 + k_2 = k$

As in Section 3, we consider three cases listed below. The main difference in the approach takes place in the first case since the most general internal LCS queries are probably hard to answer. Indeed, this query can be reduced via a binary search to $\mathcal{O}(\log n)$ two-range-LCP queries of Amir et al. [10]. With their Theorem 6, we can construct a data structure of size $\mathcal{O}(n)$ in $\mathcal{O}(n\sqrt{n})$ time that allows for $\tilde{\mathcal{O}}(\sqrt{n})$-time queries. We cannot use this data structure in our scheme though due to its high preprocessing cost. In fact, Amir et al. [10] show that the two-range-LCP data structure problem is at least as hard as the *Set Emptiness* problem: preprocess a collection of sets of total cardinality n so that queries of whether the intersection of two sets is empty can be answered efficiently. The best known $\mathcal{O}(n)$-sized data structure for this problem has $\mathcal{O}(\sqrt{n/w})$-query-time, where w is the size of the computer word. The reduction of [10] can be adapted to show that answering general internal LCS queries is at least as hard as answering Set Emptiness queries. In light of this, in the first case, we develop a different global approach to circumvent answering such queries.

1. An LCS does not contain any position (or boundary between positions) in S or T where an edit took place. As it was mentioned before, this problem probably cannot be solved efficiently in the language of k-substrings. Instead, we compute such an LCS via an inherently dynamic algorithm for the DECREMENTAL LCS problem. See Section 4.1.

2. An LCS contains at least one position where an edit operation took place in exactly one of the strings. This corresponds to the (k_1, k_2)-SUBSTRING LCS problem when an LCS contains the boundary between some substrings of exactly one of S' and T'. We compute such an LCS by combining the techniques of Section 3.2 with a sliding window approach. See Section 4.2.

3. An LCS contains at least one position where an edit operation took place in both of the strings. This corresponds to the (k_1, k_2)-SUBSTRING LCS problem when an LCS contains the boundary between some substrings in both of S' and T'. We compute such an LCS by building upon the techniques of Section 3.3 and employing efficient LCE queries for k-substrings. See Section 4.3.

4.1 Decremental LCS

We use the following convenient formulation of the problem, where the only letter that can be inserted or substituted in S (resp. in T) is $\# \notin \Sigma$, (resp. $\$ \notin \Sigma$), with $\# \neq \$$. An insertion in S in Fully Dynamic LCS corresponds to an insertion of a $\#$, while both deletions and substitutions correspond to substitutions with a $\#$. T is treated similarly. We call the problem of reporting an LCS after each such operation Decremental LCS.

We first consider the case where the sought LCS (in the fully dynamic case) is of length bounded by d; we call this problem d-Bounded-Length LCS.

Before we proceed to describe a solution to this problem we discuss how to answer LCE queries efficiently in a dynamic string. We resort to the main result of Gawrychowski et al. [30] to obtain the following lemma.

▶ **Lemma 9.** *A string S of length n can be preprocessed in $\mathcal{O}(n)$ time and space so that $k = \mathcal{O}(n)$ edit operations and $m = \mathcal{O}(n)$ lcp queries, in any order, can be processed in $\mathcal{O}(\log n)$ time each, using $\mathcal{O}(k \log n + m \log n)$ space in total.*

▶ **Lemma 10.** *d-Bounded-Length LCS can be solved in $\mathcal{O}(d \log^2 n)$ time per operation after $\tilde{\mathcal{O}}(n)$-time preprocessing, using $\tilde{\mathcal{O}}(n + kd)$ space for k performed operations.*

Proof. Let U and V be the multisets of d-length substrings and the $d-1$ suffixes of length smaller than d of S and T, respectively. We will maintain balanced BSTs B_X, with respect to the lexicographical order, containing the elements of X, for $X = U, V$, stored as substrings. We can search in these balanced BSTs in $\mathcal{O}(\log^2 n)$ since a comparison in it is an lcp query, which requires $\mathcal{O}(\log n)$ by Lemma 9, possibly followed by a character comparison. Each node of B_X will maintain a counter denoting its multiplicity in X. Let $Y = U \cup V$; we do not use Y in the algorithm, we just introduce it for conceptual convenience.

▶ **Observation 11.** *The length of the LCS of length at most d is equal to the maximum lcp between pairs of consecutive substrings in (the sorted) Y that originate from different strings.*

During preprocessing, we compute the lcp of all pairs described in Observation 11 and store them in a max heap H. To each element of the heap, we store a pointer from the nodes $u \in B_U, v \in B_V$ it originates from.

Each edit in S or T yields $\mathcal{O}(d)$ deletions and $\mathcal{O}(d)$ insertions of substrings in each of U, V and Y. We first perform deletions and then insertions. For each such operation, we have to check if it destroys or creates a pair of consecutive elements in (the sorted) Y, originating from different strings. We observe that upon the insertion/deletion of a string P, only pairs involving P, $pred_U(P)$, $pred_V(P)$, $succ_U(P)$ and $succ_V(P)$ may be involved, where $pred, succ$ are predecessor and successor with respect to the lexicographical order. These elements can be identified in $\mathcal{O}(\log^2 n)$ time. The max heap can then be updated using a constant number of LCE queries and heap updates. By Lemma 9, LCE queries (and heap updates) require $\mathcal{O}(\log n)$ time each. Finally, we return the maximum element of the heap. ◀

We now focus on the harder case that the sought LCS is of length at least d.

Let S' and T' be the strings S and T after p operations; for some $p \leq k$. For a position i, by $succ_{S'}^{\#}(i)$ we denote the smallest position $j \geq i$ such that $S'[j] = \#$. If no such position exists, we set $succ_{S'}^{\#}(i) = |S'| + 1$. Similarly, by $pred_{S'}^{\#}(i)$ we denote the greatest position $j \leq i$ such that $S'[j] = \#$, or 0 if no such position exists. Similarly we define $succ_{T'}^{\$}(i)$ and $pred_{T'}^{\$}(i)$. Such values can be computed in $\mathcal{O}(\log n)$ time if the set of replaced positions is stored in a balanced BST (note that positions of $\#$ and $\$$ can be shifted due to insertions).
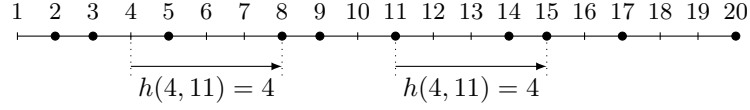
Figure 3 An example of a 6-cover $\mathbf{S}_{20}(6) = \{2, 3, 5, 8, 9, 11, 14, 15, 17, 20\}$, with the elements marked as black circles. For example, we may have $h(4, 11) = 4$ since $4 + 4, 11 + 4 \in \mathbf{S}_{20}(6)$.

We say that a set $\mathbf{S}(d) \subseteq \mathbb{Z}_+$ is a *d-cover* if there is a constant-time computable function h such that for $i, j \in \mathbb{Z}_+$ we have $0 \le h(i, j) < d$ and $i + h(i, j), j + h(i, j) \in \mathbf{S}(d)$.

▶ **Lemma 12** ([39, 15]). *For each $d \in \mathbb{Z}_+$ there is a d-cover $\mathbf{S}(d)$ such that $\mathbf{S}(d) \cap [1, n]$ is of size $\mathcal{O}(\frac{n}{\sqrt{d}})$ and can be constructed in $\mathcal{O}(\frac{n}{\sqrt{d}})$ time.*

The intuition behind applying the d-cover in our string-processing setting is as follows (inspect also Figure 3). Consider a position i on S and a position j on T. Note that $i, j \in [1, n]$. By the d-cover construction, we have that $h(i, j)$ is within distance d and $i + h(i, j), j + h(i, j) \in \mathbf{S}(d)$. Thus if we want to find a longest common substring of length *at least d*, it suffices to compute longest common extensions to the left and to the right *of only* positions $i', j' \in \mathbf{S}(d)$ (black circles in Figure 3) and then merge these partial results accordingly.

For this we use the following auxiliary problem that was introduced in [16].

TWO STRING FAMILIES LCP
Input: A compact trie $\mathcal{T}(\mathcal{F})$ of a family of strings $\mathcal{F}$ and two sets $\mathcal{P}, \mathcal{Q} \subseteq \mathcal{F}^2$
Output: The value maxPairLCP$(\mathcal{P}, \mathcal{Q})$, defined as
maxPairLCP$(\mathcal{P}, \mathcal{Q}) = \max\{\mathsf{lcp}(P_1, Q_1) + \mathsf{lcp}(P_2, Q_2) : (P_1, P_2) \in \mathcal{P} \text{ and } (Q_1, Q_2) \in \mathcal{Q}\}$

An efficient solution to this problem was shown in [16] (and, implicitly, in [20, 25]).

▶ **Lemma 13** ([16]). TWO STRING FAMILIES LCP *can be solved in $\mathcal{O}(|\mathcal{F}| + N \log N)$ time, where $N = |\mathcal{P}| + |\mathcal{Q}|$.*

▶ **Lemma 14.** DECREMENTAL LCS *can be solved in $\tilde{\mathcal{O}}(n^{2/3})$ time per query, using $\tilde{\mathcal{O}}(n + kn^{2/3})$ space, after $\tilde{\mathcal{O}}(n)$-time preprocessing for k performed operations.*

Proof. Let us consider an integer $d \in [1, n]$. For lengths up to d, we use the algorithm for the BOUNDED-LENGTH LCS problem of Lemma 10. If this problem indicates that there is a solution of length at least d, we proceed to the second step. Let $A = \mathbf{S}(d) \cap [1, n]$ be a d-cover of size $\mathcal{O}(n/\sqrt{d})$ (see Lemma 12).

We consider the following families of pairs of strings: $\mathcal{P} = \{ (S[pred_{S'}^{\#}(i-1) + 1 .. i - 1])^R, S[i .. succ_{S'}^{\#}(i) - 1]) : i \in A \}$ and $\mathcal{Q} = \{ (T[pred_{T'}^{\$}(i-1) + 1 .. i - 1])^R, T[i .. succ_{T'}^{\$}(i) - 1]) : i \in A\}$. We define $\mathcal{F}$ as the family of strings that occur in the pairs from $\mathcal{P}$ and $\mathcal{Q}$. Then maxPairLCP$(\mathcal{P}, \mathcal{Q})$ equals the length of the sought LCS, provided that it is at least d.

Note that $|\mathcal{P}|, |\mathcal{Q}|, |\mathcal{F}|$ are $\mathcal{O}(n/\sqrt{d})$. A compact trie $\mathcal{T}(\mathcal{F})$ can be constructed in $\mathcal{O}(|\mathcal{F}| \log |\mathcal{F}|)$ time by sorting all the strings (using lcp-queries) and then a standard left-to-right construction; see [19]. Thus we can use the solution to TWO STRING FAMILIES LCP which takes $\tilde{\mathcal{O}}(n/\sqrt{d})$ time. We set $d = \lfloor n^{2/3} \rfloor$ to obtain the stated complexity. ◀

4.2 One-Sided Cross-Substring Queries

We show a solution with $\tilde{\mathcal{O}}(k^2)$-time queries after $\tilde{\mathcal{O}}(n)$-time preprocessing by building upon the techniques from Section 3.2.
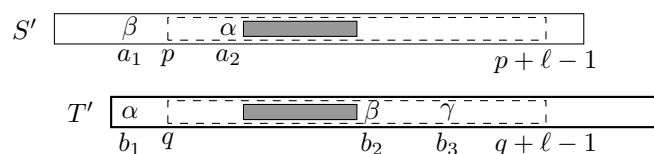
Figure 4 Occurrences of an LCS of S' and T' crossing the boundaries in both are denoted by dashed rectangles. The starting positions f_{i+1} and g_{j+1} minimize the formula $|(f_{i+1} - p) - (g_{j+1} - q)|$. Hence the gray rectangle, denoting U, is a prefix of $S'[f_{i+1} \ldots f_{i+2} - 1]$ and a suffix of $T'[g_j \ldots g_{j+1} - 1]$. We thus process it as a border while processing (F_i, F_{i+1}) and (G_j, G_{j+1}) and hence find this LCS.

We present an algorithm that computes, for each $i = 1, \ldots, k_1$, the longest substring of S' that contains the first character of F_i, but not of F_{i-1} and occurs in G_p for a given $p \in \{1, \ldots, k_2\}$ in $\tilde{\mathcal{O}}(k)$ time. These are the possible LCSs that cross the substring boundaries.

Let us start by a global part of the computation. For convenience let us assume that $F_0 = F_{k_1+1}$ are empty strings. For an index $i \in \{1, \ldots, k_1\}$, by $next(i)$ we denote the greatest index $j \geq i - 1$ for which $F_i \ldots F_j$ is a substring of T. These values are computed using a sliding-window-based approach. We start with computing $next(1)$. To this end, we use Lemma 5(a) for subsequent substrings $F_1, F_2, \ldots$ as long as their concatenation is a substring of T. This takes $\mathcal{O}(k \log \log n)$ time. Now assume that we have computed $next(i)$ and we wish to compute $next(i+1)$. Obviously, $next(i+1) \geq next(i)$. Let $j = next(i)$. We start with $F_{i+1} \ldots F_j$ which is represented as a substring of T. We keep extending this substring by $F_{j+1}, F_{j+2}, \ldots$ using Lemma 5(a) as before as long as the concatenation is a substring of T. In total, computing values $next(i)$ for all $i = 1, \ldots, k_1$ takes $\tilde{\mathcal{O}}(k)$ time.

Let us now fix i and let $j = next(i)$. We use Lemma 5(b) to find the longest prefix P_i of $(F_i \ldots F_j)F_{j+1}$ that occurs in T; it is also the longest prefix of $F_i \ldots F_{k_1}$ that occurs in T by the definition of $next(i)$. Then Lemma 5(b) can be used to compute the longest suffix Q_i of F_{i-1} that occurs in T. For each i it takes time $\tilde{\mathcal{O}}(1)$ time to find P_i and Q_i after $\tilde{\mathcal{O}}(n)$ time and space preprocessing.

We then compute the sought result for given $i \in \{1, \ldots, k_1\}$ and $p \in \{1, \ldots, k_2\}$ by a THREE SUBSTRINGS LCS query for $U = Q_i$, $V = P_i$, and $W = G_p$. With Lemma 6 this takes $\tilde{\mathcal{O}}(k^2)$ time in total after $\tilde{\mathcal{O}}(n)$ time and space preprocessing.

4.3   Two-Sided Cross-Substring Queries

We show a solution with $\mathcal{O}(k^2 \log^3 n)$-time queries after $\mathcal{O}(n \log n)$-time preprocessing by combining the ideas presented in Section 3.3 and efficient LCE queries in the dynamic setting (cf. Lemma 9). We consider each pair of boundaries between pairs (F_i, F_{i+1}) and (G_j, G_{j+1}), for $1 \leq i \leq k_1 - 1$ and $1 \leq j \leq k_2 - 1$. We process the prefixes of F_{i+1} that are suffixes of G_j as in Section 3.3 (the symmetric case is treated analogously).

We next argue that we do not miss any possible LCS by only considering such prefix-suffix pairs of F_{i+1} and G_j. Let f_i and g_i be the starting positions of F_i and G_j in S' and T', respectively. An LCS $S'[p \ldots p + \ell - 1] = T'[q \ldots q + \ell - 1]$ of this type will be reported when processing the pairs (F_i, F_{i+1}) and (G_j, G_{j+1}), satisfying $p \leq f_{i+1} \leq p + \ell - 1$, $q \leq g_{j+1} \leq q + \ell - 1$, for which $|(f_{i+1} - p) - (g_{j+1} - q)|$ is minimal. Without loss of generality assume $f_{i+1} - p \leq g_{j+1} - q$. Then, $S'[f_{i+1} \ldots p + g_{j+1} - q - 1]$ is a prefix of F_{i+1} and $T'[q + f_{j+1} - p + 1 \ldots g_{j+1} - 1]$ is a suffix of G_j and hence it is a prefix-suffix that will be processed by our algorithm; see Figure 4.

We assume that $k = \mathcal{O}(\sqrt{n})$, which is sufficient for our main result. We consider $k_1 \times k_2 = \mathcal{O}(k^2)$ pairs $(F_i, F_{i+1}), (G_j, G_{j+1})$ and, by the analysis in Section 3.3, the time required for processing each of them (i.e. finding the longest prefix-suffix and then considering all its borders in $\mathcal{O}(\log n)$ batches) is bounded by the time required to answer $\mathcal{O}(\log n)$ LCE queries, which can be answered in time $\mathcal{O}(k^2 \log n)$ by Lemma 9. By Lemma 9 we have that $k^2 = \mathcal{O}(n)$ LCE queries over a k-substring of S, can be performed in $\mathcal{O}(k^2 \log n)$ time, using this much extra space, after $\mathcal{O}(n)$-time preprocessing. Hence the total time required is $\mathcal{O}(k^2 \log^3 n)$ after $\tilde{\mathcal{O}}(n)$-time preprocessing.

4.4   Main Result

By combining the results of Sections 4.1 to 4.3 we obtain the following.

▶ **Lemma 15.** (k_1, k_2)-SUBSTRING LCS *queries can be answered in* $\tilde{\mathcal{O}}(n^{2/3} + k^2)$ *time, where* $k = k_1 + k_2 = \mathcal{O}(\sqrt{n})$, *using a data structure that can be constructed in* $\tilde{\mathcal{O}}(n)$ *time.*

We now formalize the time slicing deamortization technique for our purposes.

▶ **Lemma 16.** *Assume that there is a data structure* $\mathcal{D}$ *over an input string of length* n *that occupies* $\mathcal{O}(s_n)$ *space, answers queries for* k-*substrings in time* $\mathcal{O}(q_n(k))$ *and can be constructed in time* $\mathcal{O}(t_n)$. *Assume that* $s_n \geq n$ *and* $q(k, n) \geq k$ *is non-decreasing with respect to* k. *We can then design an algorithm that preprocesses the input string in time* $\mathcal{O}(t_n)$ *and answers queries dynamically under edit operations in worst-case time* $\tilde{\mathcal{O}}(q_n(\kappa))$, *where* κ *is such that* $q_n(\kappa) = (t_n + n)/\kappa$, *using* $\mathcal{O}(s_n)$ *space.*

By plugging Lemma 15 into Lemma 16 we arrive at our main result.

▶ **Theorem 17.** FULLY DYNAMIC LCS *on two strings, each of length up to* n, *can be solved in* $\tilde{\mathcal{O}}(n^{2/3})$ *time per operation, using* $\tilde{\mathcal{O}}(n)$ *space, after* $\tilde{\mathcal{O}}(n)$-*time preprocessing.*

5   Applications

We present three applications of our techniques. The fully dynamic algorithm for computing the longest repeat is very similar to the fully dynamic algorithm for LCS.

Another application is a fully dynamic algorithm for the longest palindrome substring which extends the results of [26, 27]. We consider two cases. In the internal case, in which the longest palindrome occurs between edited positions, we use range queries on the set of maximal palindrome substrings of a string (which is known to have linear size). In the cross-substring case, we use the known fact that the lengths of suffix palindromes of a string can be represented as a logarithmic number of arithmetic progressions which lets us use string periodicity similarly as in Section 3.3. We remark that a more efficient algorithm for computing the longest palindrome in a dynamic string has recently been proposed [6].

The authors of [48] presented algorithms for computing a representation of a Lyndon factorization of a prefix of a string and of a suffix of a string in $\tilde{\mathcal{O}}(1)$ time after $\tilde{\mathcal{O}}(n)$ preprocessing. For the prefixes, their solution is based on the Lyndon representations of prefixes of a Lyndon string, whereas for the suffixes, it is based on the structure of a Lyndon tree (originally due to [13]). In order to devise our fully dynamic algorithm, we carefully combine these two approaches to obtain general internal computation of a representation of a Lyndon factorization in the same time bounds.

6 Final Remarks

We anticipate that the techniques presented in this paper to obtain fully dynamic algorithms for several classical problems on strings are applicable in a wider range of problems on strings.

The significance of our results is additionally highlighted by the following argument. It is known that finding an LCS when the strings have wildcard characters [2] or when $k = \Omega(\log n)$ mismatches are allowed [36] in strongly subquadratic time would refute the Strong Exponential Time Hypothesis (SETH) [33, 32] (on the other hand, pattern matching with wildcard characters can be solved in $\tilde{\mathcal{O}}(n)$ time [17] and with k mismatches in $\tilde{\mathcal{O}}(n\sqrt{k})$ time [9]). It is therefore unlikely that a fully dynamic algorithm with strongly sublinear-time queries exists for these problems: such an algorithm could be trivially applied as a black box to solve the problems in their static setting in strongly subquadratic time, refuting SETH.

This research could inspire more work on the lower bound side of dynamic problems. The currently known hardness (and conditional hardness) results for dynamic problems on strings have been established for dynamic pattern matching [18, 30]. It would be interesting to investigate (conditional) lower bounds for the dynamic problems considered in this paper.

References

1 Amir Abboud, Richard Ryan Williams, and Huacheng Yu. More Applications of the Polynomial Method to Algorithm Design. In Piotr Indyk, editor, *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015, San Diego, CA, USA, January 4-6, 2015*, pages 218–230. SIAM, 2015. `doi:10.1137/1.9781611973730.17`.

2 Amir Abboud, Virginia Vassilevska Williams, and Oren Weimann. Consequences of Faster Alignment of Sequences. In Javier Esparza, Pierre Fraigniaud, Thore Husfeldt, and Elias Koutsoupias, editors, *Automata, Languages, and Programming - 41st International Colloquium, ICALP 2014, Copenhagen, Denmark, July 8-11, 2014, Proceedings, Part I*, volume 8572 of *Lecture Notes in Computer Science*, pages 39–51. Springer, 2014. `doi:10.1007/978-3-662-43948-7_4`.

3 Paniz Abedin, Sahar Hooshmand, Arnab Ganguly, and Sharma V. Thankachan. The Heaviest Induced Ancestors Problem Revisited. In Gonzalo Navarro, David Sankoff, and Binhai Zhu, editors, *Annual Symposium on Combinatorial Pattern Matching, CPM 2018, July 2-4, 2018 - Qingdao, China*, volume 105 of *LIPIcs*, pages 20:1–20:13. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018. `doi:10.4230/LIPIcs.CPM.2018.20`.

4 Stephen Alstrup, Gerth Stølting Brodal, and Theis Rauhe. Pattern Matching in Dynamic Texts. In *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '00, pages 819–828, Philadelphia, PA, USA, 2000. Society for Industrial and Applied Mathematics. URL: `http://dl.acm.org/citation.cfm?id=338219.338645`.

5 Amihood Amir and Itai Boneh. Locally Maximal Common Factors as a Tool for Efficient Dynamic String Algorithms. In Gonzalo Navarro, David Sankoff, and Binhai Zhu, editors, *Annual Symposium on Combinatorial Pattern Matching, CPM 2018, July 2-4, 2018 - Qingdao, China*, volume 105 of *LIPIcs*, pages 11:1–11:13. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018. `doi:10.4230/LIPIcs.CPM.2018.11`.

6 Amihood Amir and Itai Boneh. Dynamic Palindrome Detection. *CoRR*, abs/1906.09732, 2019. `arXiv:1906.09732`.

7 Amihood Amir, Itai Boneh, Panagiotis Charalampopoulos, and Eitan Kondratovsky. Repetition Detection in a Dynamic String. In Michael A. Bender, Ola Svensson, and Grzegorz Herman, editors, *27th Annual European Symposium on Algorithms, ESA 2019, September 9-13, 2019, Munich, Germany*, LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2019.

8   Amihood Amir, Panagiotis Charalampopoulos, Costas S. Iliopoulos, Solon P. Pissis, and Jakub Radoszewski. Longest Common Factor After One Edit Operation. In Gabriele Fici, Marinella Sciortino, and Rossano Venturini, editors, *String Processing and Information Retrieval: 24th International Symposium, SPIRE 2017, Palermo, Italy, September 26–29, 2017, Proceedings*, volume 10508 of *Lecture Notes in Computer Science*, pages 14–26. Springer International Publishing, 2017. `doi:10.1007/978-3-319-67428-5_2`.

9   Amihood Amir, Moshe Lewenstein, and Ely Porat. Faster algorithms for string matching with k mismatches. *Journal of Algorithms*, 50(2):257–275, 2004. `doi:10.1016/S0196-6774(03)00097-X`.

10   Amihood Amir, Moshe Lewenstein, and Sharma V. Thankachan. Range LCP Queries Revisited. In Costas S. Iliopoulos, Simon J. Puglisi, and Emine Yilmaz, editors, *String Processing and Information Retrieval - 22nd International Symposium, SPIRE 2015, London, UK, September 1-4, 2015, Proceedings*, volume 9309 of *Lecture Notes in Computer Science*, pages 350–361. Springer, 2015. `doi:10.1007/978-3-319-23826-5_33`.

11   Alberto Apostolico, Maxime Crochemore, Martin Farach-Colton, Zvi Galil, and S. Muthukrishnan. Forty Years of Text Indexing. In Johannes Fischer and Peter Sanders, editors, *Combinatorial Pattern Matching, 24th Annual Symposium, CPM 2013, Bad Herrenalb, Germany, June 17-19, 2013. Proceedings*, volume 7922 of *Lecture Notes in Computer Science*, pages 1–10. Springer, 2013. `doi:10.1007/978-3-642-38905-4_1`.

12   Lorraine A. K. Ayad, Carl Barton, Panagiotis Charalampopoulos, Costas S. Iliopoulos, and Solon P. Pissis. Longest Common Prefixes with k-Errors and Applications. In Travis Gagie, Alistair Moffat, Gonzalo Navarro, and Ernesto Cuadros-Vargas, editors, *String Processing and Information Retrieval - 25th International Symposium, SPIRE 2018, Lima, Peru, October 9-11, 2018, Proceedings*, volume 11147 of *Lecture Notes in Computer Science*, pages 27–41. Springer, 2018. `doi:10.1007/978-3-030-00479-8_3`.

13   Hélène Barcelo. On the action of the symmetric group on the Free Lie Algebra and the partition lattice. *Journal of Combinatorial Theory, Series A*, 55(1):93–129, 1990. `doi:10.1016/0097-3165(90)90050-7`.

14   Michael A. Bender and Martin Farach-Colton. The LCA Problem Revisited. In Gaston H. Gonnet, Daniel Panario, and Alfredo Viola, editors, *LATIN 2000: Theoretical Informatics, 4th Latin American Symposium, Punta del Este, Uruguay, April 10-14, 2000, Proceedings*, volume 1776 of *Lecture Notes in Computer Science*, pages 88–94. Springer, 2000. `doi:10.1007/10719839_9`.

15   Stefan Burkhardt and Juha Kärkkäinen. Fast Lightweight Suffix Array Construction and Checking. In Ricardo A. Baeza-Yates, Edgar Chávez, and Maxime Crochemore, editors, *Combinatorial Pattern Matching, CPM 2003, Morelia, Michocán, Mexico, June 25–27, 2003*, volume 2676 of *Lecture Notes in Computer Science*, pages 55–69. Springer, 2003. `doi:10.1007/3-540-44888-8_5`.

16   Panagiotis Charalampopoulos, Maxime Crochemore, Costas S. Iliopoulos, Tomasz Kociumaka, Solon P. Pissis, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. Linear-Time Algorithm for Long LCF with k Mismatches. In Gonzalo Navarro, David Sankoff, and Binhai Zhu, editors, *Annual Symposium on Combinatorial Pattern Matching, CPM 2018, July 2-4, 2018 - Qingdao, China*, volume 105 of *LIPIcs*, pages 23:1–23:16. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018. `doi:10.4230/LIPIcs.CPM.2018.23`.

17   Peter Clifford and Raphaël Clifford. Simple deterministic wildcard matching. *Information Processing Letters*, 101(2):53–54, 2007. `doi:10.1016/j.ipl.2006.08.002`.

18   Raphaël Clifford, Allan Grønlund, Kasper Green Larsen, and Tatiana A. Starikovskaya. Upper and Lower Bounds for Dynamic Data Structures on Strings. In Rolf Niedermeier and Brigitte Vallée, editors, *35th Symposium on Theoretical Aspects of Computer Science, STACS 2018, February 28 to March 3, 2018, Caen, France*, volume 96 of *LIPIcs*, pages 22:1–22:14. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018. `doi:10.4230/LIPIcs.STACS.2018.22`.

19 Maxime Crochemore, Christophe Hancart, and Thierry Lecroq. *Algorithms on Strings*. Cambridge University Press, 2007.

20 Maxime Crochemore, Costas S. Iliopoulos, Manal Mohamed, and Marie-France Sagot. Longest repeats with a block of k don't cares. *Theoretical Computer Science*, 362(1-3):248–254, 2006. `doi:10.1016/j.tcs.2006.06.029`.

21 Martin Farach. Optimal Suffix Tree Construction with Large Alphabets. In *38th Annual Symposium on Foundations of Computer Science, FOCS '97, Miami Beach, Florida, USA, October 19-22, 1997*, pages 137–143. IEEE Computer Society, 1997. `doi:10.1109/SFCS.1997.646102`.

22 Paolo Ferragina. Dynamic Text Indexing under String Updates. *Journal of Algorithms*, 22(2):296–328, 1997. `doi:10.1006/jagm.1996.0814`.

23 Paolo Ferragina and Roberto Grossi. Optimal On-Line Search and Sublinear Time Update in String Matching. *SIAM Journal on Computing*, 27(3):713–736, 1998. `doi:10.1137/S0097539795286119`.

24 Johannes Fischer, Dominik Köppl, and Florian Kurpicz. On the Benefit of Merging Suffix Array Intervals for Parallel Pattern Matching. In Roberto Grossi and Moshe Lewenstein, editors, *27th Annual Symposium on Combinatorial Pattern Matching, CPM 2016, June 27-29, 2016, Tel Aviv, Israel*, volume 54 of *LIPIcs*, pages 26:1–26:11. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016. `doi:10.4230/LIPIcs.CPM.2016.26`.

25 Tomás Flouri, Emanuele Giaquinta, Kassian Kobert, and Esko Ukkonen. Longest common substrings with k mismatches. *Information Processing Letters*, 115(6-8):643–647, 2015.

26 Mitsuru Funakoshi, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Longest substring palindrome after edit. In Gonzalo Navarro, David Sankoff, and Binhai Zhu, editors, *Annual Symposium on Combinatorial Pattern Matching, CPM 2018, July 2–4, 2018 – Qingdao, China*, volume 105 of *LIPIcs*, pages 12:1–12:14. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2018. `doi:10.4230/LIPIcs.CPM.2018.12`.

27 Mitsuru Funakoshi, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Faster Queries for Longest Substring Palindrome After Block Edit. In Nadia Pisanti and Solon P. Pissis, editors, *30th Annual Symposium on Combinatorial Pattern Matching, CPM 2019, June 18-20, 2019, Pisa, Italy*, volume 128 of *LIPIcs*, pages 27:1–27:13. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2019. `doi:10.4230/LIPIcs.CPM.2019.27`.

28 Travis Gagie, Paweł Gawrychowski, and Yakov Nekrich. Heaviest Induced Ancestors and Longest Common Substrings. In *Proceedings of the 25th Canadian Conference on Computational Geometry, CCCG 2013, Waterloo, Ontario, Canada, August 8-10, 2013*. Carleton University, Ottawa, Canada, 2013. URL: `http://cccg.ca/proceedings/2013/papers/paper_29.pdf`.

29 Paweł Gawrychowski, Adam Karczmarz, Tomasz Kociumaka, Jakub Łącki, and Piotr Sankowski. Optimal Dynamic Strings. *CoRR*, abs/1511.02612, 2015. `arXiv:1511.02612`.

30 Paweł Gawrychowski, Adam Karczmarz, Tomasz Kociumaka, Jakub Łącki, and Piotr Sankowski. Optimal Dynamic Strings. In Artur Czumaj, editor, *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, New Orleans, LA, USA, January 7-10, 2018*, pages 1509–1528. SIAM, 2018. `doi:10.1137/1.9781611975031.99`.

31 Ming Gu, Martin Farach, and Richard Beigel. An Efficient Algorithm for Dynamic Text Indexing. In *Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '94, pages 697–704, Philadelphia, PA, USA, 1994. Society for Industrial and Applied Mathematics. URL: `http://dl.acm.org/citation.cfm?id=314464.314675`.

32 Russell Impagliazzo and Ramamohan Paturi. On the Complexity of k-SAT. *Journal of Computer and System Sciences*, 62(2):367–375, 2001. `doi:10.1006/jcss.2000.1727`.

33 Russell Impagliazzo, Ramamohan Paturi, and Francis Zane. Which Problems Have Strongly Exponential Complexity? *Journal of Computer and System Sciences*, 63(4):512–530, December 2001. `doi:10.1006/jcss.2001.1774`.

34   Tomasz Kociumaka. *Efficient Data Structures for Internal Queries in Texts*. PhD thesis, University of Warsaw, October 2018. URL: `https://www.mimuw.edu.pl/~kociumaka/files/phd.pdf`.

35   Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. Internal Pattern Matching Queries in a Text and Applications. In Piotr Indyk, editor, *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015, San Diego, CA, USA, January 4-6, 2015*, pages 532–551. SIAM, 2015. `doi:10.1137/1.9781611973730.36`.

36   Tomasz Kociumaka, Jakub Radoszewski, and Tatiana A. Starikovskaya. Longest Common Substring with Approximately k Mismatches. *Algorithmica*, 81(6):2633–2652, 2019. `doi:10.1007/s00453-019-00548-x`.

37   Tomasz Kociumaka, Tatiana A. Starikovskaya, and Hjalte Wedel Vildhøj. Sublinear Space Algorithms for the Longest Common Substring Problem. In Andreas S. Schulz and Dorothea Wagner, editors, *Algorithms - ESA 2014 - 22th Annual European Symposium, Wroclaw, Poland, September 8-10, 2014. Proceedings*, volume 8737 of *Lecture Notes in Computer Science*, pages 605–617. Springer, 2014. `doi:10.1007/978-3-662-44777-2_50`.

38   Roger C. Lyndon. On Burnside's problem. *Transactions of the American Mathematical Society*, 77:202–215, 1954.

39   Mamoru Maekawa. A $\sqrt{n}$ Algorithm for Mutual Exclusion in Decentralized Systems. *ACM Transactions on Computer Systems*, 3(2):145–159, 1985. `doi:10.1145/214438.214445`.

40   Kurt Mehlhorn, R. Sundar, and Christian Uhrig. Maintaining Dynamic Sequences under Equality Tests in Polylogarithmic Time. *Algorithmica*, 17(2):183–198, 1997. `doi:10.1007/BF02522825`.

41   Süleyman Cenk Sahinalp and Uzi Vishkin. Efficient Approximate and Dynamic Matching of Patterns Using a Labeling Paradigm (extended abstract). In *37th Annual Symposium on Foundations of Computer Science, FOCS '96, Burlington, Vermont, USA, 14-16 October, 1996*, pages 320–328. IEEE Computer Society, 1996. `doi:10.1109/SFCS.1996.548491`.

42   Daniel D. Sleator and Robert Endre Tarjan. A Data Structure for Dynamic Trees. *Journal of Computer and System Sciences*, 26(3):362–391, June 1983. `doi:10.1016/0022-0000(83)90006-5`.

43   Tatiana A. Starikovskaya. Longest Common Substring with Approximately k Mismatches. In Roberto Grossi and Moshe Lewenstein, editors, *27th Annual Symposium on Combinatorial Pattern Matching, CPM 2016, June 27-29, 2016, Tel Aviv, Israel*, volume 54 of *LIPIcs*, pages 21:1–21:11. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016. `doi:10.4230/LIPIcs.CPM.2016.21`.

44   Tatiana A. Starikovskaya and Hjalte Wedel Vildhøj. Time-Space Trade-Offs for the Longest Common Substring Problem. In Johannes Fischer and Peter Sanders, editors, *Combinatorial Pattern Matching, 24th Annual Symposium, CPM 2013, Bad Herrenalb, Germany, June 17-19, 2013. Proceedings*, volume 7922 of *Lecture Notes in Computer Science*, pages 223–234. Springer, 2013. `doi:10.1007/978-3-642-38905-4_22`.

45   Rajamani Sundar and Robert Endre Tarjan. Unique Binary-Search-Tree Representations and Equality Testing of Sets and Sequences. *SIAM Journal on Computing*, 23(1):24–44, 1994. `doi:10.1137/S0097539790189733`.

46   Sharma V. Thankachan, Chaitanya Aluru, Sriram P. Chockalingam, and Srinivas Aluru. Algorithmic Framework for Approximate Matching Under Bounded Edits with Applications to Sequence Analysis. In Benjamin J. Raphael, editor, *Research in Computational Molecular Biology - 22nd Annual International Conference, RECOMB 2018, Paris, France, April 21-24, 2018, Proceedings*, volume 10812 of *Lecture Notes in Computer Science*, pages 211–224. Springer, 2018. `doi:10.1007/978-3-319-89929-9_14`.

47   Sharma V. Thankachan, Alberto Apostolico, and Srinivas Aluru. A Provably Efficient Algorithm for the k-Mismatch Average Common Substring Problem. *Journal of Computational Biology*, 23(6):472–482, 2016. `doi:10.1089/cmb.2015.0235`.

48   Yuki Urabe, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Longest Lyndon Substring After Edit. In Gonzalo Navarro, David Sankoff, and Binhai Zhu, editors, *Annual Symposium on Combinatorial Pattern Matching, CPM 2018, July 2-4, 2018 – Qingdao, China*, volume 105 of *LIPIcs*, pages 19:1–19:10. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018. `doi:10.4230/LIPIcs.CPM.2018.19`.

49   Peter Weiner. Linear Pattern Matching Algorithms. In *14th Annual Symposium on Switching and Automata Theory, Iowa City, Iowa, USA, October 15-17, 1973*, pages 1–11. IEEE Computer Society, 1973. `doi:10.1109/SWAT.1973.13`.