Algorithmica (2020) 82:3707–3743
https://doi.org/10.1007/s00453-020-00744-0

Dynamic and Internal Longest Common Substring

Amihood Amir[1] · Panagiotis Charalampopoulos[2] · Solon P. Pissis[3,4] · Jakub Radoszewski[5,6]

Received: 14 January 2020 / Accepted: 25 June 2020 / Published online: 15 July 2020

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 the fully dynamic LCS 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. This line of research has been recently initiated in a somewhat restricted dynamic variant by Amir et al. [SPIRE 2017]. More specifically, the authors 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; specifically, computing the longest palindrome and the Lyndon factorization of a string after a single edit operation. We develop dynamic sublinear-time algorithms for both of these problems as well. We also consider *internal* LCS queries, that is, queries in which we are to return an LCS of a pair of substrings of S and T. We show that answering such queries is hard in general and propose efficient data structures for several restricted cases.

Keywords Longest common substring · String algorithms · Dynamic algorithms

A preliminary version of this paper was presented at the 27th Annual European Symposium on Algorithms (ESA 2019) [12].

The $\tilde{\mathcal{O}}(\cdot)$ notation suppresses $\log^{\mathcal{O}(1)} n$ factors for inputs of size n

✉ Solon P. Pissis
solon.pissis@cwi.nl

Extended author information available on the last page of the article

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 is in $\Omega(n \log n)$. In 1973 Weiner solved it in the optimal $\mathcal{O}(n)$ time [78] designing a data structure that was later called the suffix tree (see also [41]). Knuth declared Weiner's algorithm the "Algorithm of the Year" [16]. Since $\mathcal{O}(n)$ time is optimal for this problem, a series of studies have been dedicated in improving the working space [63, 73]. The LCS problem has also been studied under Hamming and edit distance. We refer the interested reader to [2, 17, 28, 72, 75, 76] and references therein.

In [72], 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 $S[4] := \mathtt{a}$ is performed. The next substitution, $T[3] := \mathtt{b}$, *halves* the length of an LCS.

$$\begin{array}{ccccc} S = \mathtt{c\underline{aab}aaa} & S[4] := \mathtt{a} & S = \mathtt{c\underline{aaaaaa}} & T[3] := \mathtt{b} & S = \mathtt{c\underline{aaa}aaa} \\ T = \mathtt{aaaa\underline{aab}} & & T = \mathtt{\underline{aaaaaa}b} & & T = \mathtt{aab\underline{aaa}b} \end{array}$$

Amir et al. [11] introduced a restricted dynamic variant, where any *single* edit operation is allowed and is reverted afterwards. We call this problem LCS AFTER ONE EDIT. They 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. Abedin et al. [3] improved the complexities of this data structure by $\log^{\mathcal{O}(1)} n$ factors. Two other restricted variants of the dynamic LCS problem were considered by Amir and Boneh in [8]. 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.

This work initiated a new line of research on analogously restricted dynamic variants of problems on strings. 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 [64]. Computing a longest palindrome and a longest Lyndon substring of a string after a single edit have been recently studied in [48] (see also [49]) and in [77], respectively.

In this paper we make substantial progress: we show a strongly sublinear-time solution for the general version of the LCS 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. We also develop fully dynamic sublinear-time algorithms for computing a longest palindrome and for maintaining the Lyndon factorization of a string.

Below we mention some known results on dynamic and internal 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 [41, 78]. 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 [42, 43, 53]. The first data structure with polylogarithmic update time and time-optimal queries was shown by Sahinalp and Vishkin [70]. The update time was later improved by Alstrup et al. [7] at the expense of slightly suboptimal query time. The state of the art is the data structure by Gawrychowski et al. [51] supporting time-optimal queries with $\mathcal{O}(\log^2 n)$ time for updates. Clifford et al. [35] 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 $\mathbf{W}$ of strings of total length n supporting the following operations: adding a string to $\mathbf{W}$, adding the concatenation of two strings from $\mathbf{W}$ to $\mathbf{W}$, splitting a string from $\mathbf{W}$ and adding the two residual strings in $\mathbf{W}$, and returning the length of the longest common prefix of two strings from $\mathbf{W}$. This line of research was initiated by Sundar and Tarjan [74]. Data structures supporting updates in polylogarithmic time were presented by Mehlhorn et al. [67] and Alstrup et al. [7]. Finally, Gawrychowski et al. [51] proposed an optimal solution with $\mathcal{O}(\log n)$-time updates, where n is the total length of all strings in the collection. Charalampopoulos et al. [33] recently presented efficient algorithms for approximate pattern matching, under both the Hamming and edit distances, over such a collection of strings, maintained dynamically as in [51].

Dynamic Maintenance of Repetitions Squares are strings of the form XX. In [10], 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.

Dynamic String Alignment A dynamic version of the string alignment problem, which is a generalization of the well-known longest common subsequence problem, was recently studied in [32] (see also [55] for a practical algorithm). The authors showed that when the alignment weights are integers bounded in absolute value by some $w = n^{\mathcal{O}(1)}$, an optimal string alignment can be maintained in $\tilde{\mathcal{O}}(n \cdot \min\{\sqrt{n}, w\})$ time per update. This is conditionally optimal on SETH for constant w—up to polylogarithmic factors—due to the lower bounds on computing such an alignment in the static case [1, 18, 25].

Internal Pattern Matching In the so-called *internal* model—the name was coined by Kociumaka et al. in [62]—one is to answer queries about substrings of a given text. Problems that have been studied in this model include pattern matching, longest common prefix, periodicity, minimal lexicographic rotation and dictionary matching; see [61] for an overview and [30, 31, 60, 62]. We explore internal queries relevant to the considered dynamic problems and in several cases we use them as building blocks in our dynamic algorithms.

Our Results and Techniques We make the following contributions:

1. A study of internal LCS queries: hardness of the general case conditional on the hardness of set disjointness and efficient data structures for useful restricted cases, based on ingredients such as the suffix tree, heavy-path decomposition and orthogonal range queries (see Sect. 3).
2. An efficient data structure for a natural generalization of the LCS after One Edit, where we allow one edit in each of the strings. This solution relies on internal LCS queries and string periodicity (see Sect. 4).
3. 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. We employ small difference covers in order to decompose our problem in the cases of the LCS being short or long and treat each case separately (see Sect. 5).[1] We show that a simple modification to our algorithm can maintain a longest repeat of a string S of length n in the same complexities (see Sect. 6).
4. A general scheme for dynamic problems on strings. This scheme relies on having efficient data structures for the (static) internal counterparts (see Sect. 7). We show two following applications of this scheme.
5. A fully dynamic algorithm for computing a longest palindrome substring of a string S requiring $\tilde{\mathcal{O}}(\sqrt{n})$ time per edit.[2] We use the facts that the set of maximal palindrome substrings of a string has a linear size and that the lengths of suffix palindromes of a string can be represented as a logarithmic number of arithmetic progressions (see Sect. 8).
6. 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. The authors of [77] 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. We carefully combine these two representations to obtain general internal computation of a representation of a Lyndon factorization in the same time bounds (see Sect. 9).

[1] For a discussion of recent developments on this problem [29], see Sect. 10.

[2] A more efficient algorithm for computing the longest palindrome in a dynamic string has recently been proposed [9].

A preliminary version of this paper appeared in [12]. Here, in particular, we greatly simplify the algorithm for the fully dynamic LCS problem, we provide a more detailed study of internal LCS queries and we include sections and proofs missing from the preliminary version due to space constraints.

2 Preliminaries

Strings Let $S = S[1]S[2] \dots S[n]$ be a *string* of length $|S| = n$ over an integer alphabet $\Sigma = \{1, \dots, n^{\mathcal{O}(1)}\}$. The elements of Σ are called *characters*. For two positions i and j on S, we denote by $S[i\mathinner{.\,.}j] = S[i] \dots S[j]$ 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 $S[1\mathinner{.\,.}j]$ is denoted by $S^{(j)}$ and a suffix $S[i\mathinner{.\,.}n]$ 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 $S^R = S[n]S[n-1] \dots S[1]$. 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 $i = 1, \dots, n-p$. 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.

Suffix Tree The *suffix tree* $\mathcal{T}(S)$ of a string S of length n is a compact trie representing all suffixes of S. The *branching* nodes of the trie as well as the *terminal* nodes, that correspond to suffixes of S, become *explicit* nodes of the suffix tree, while the other nodes are *implicit*. Each edge of the suffix tree can be viewed as an upward maximal path of implicit nodes starting with an explicit node. Moreover, each node belongs to a unique path of that kind. Thus, each node of the trie can be represented in the suffix tree by the edge it belongs to and an index within the corresponding path. We let $\mathcal{L}(v)$ denote the *path-label* of a node v, i.e., the concatenation of the edge labels along the path from the root to v. We say that v is path-labelled $\mathcal{L}(v)$. Additionally, $\mathcal{D}(v) = |\mathcal{L}(v)|$ is used to denote the *string-depth* of node v. A terminal node v such that $\mathcal{L}(v) = S_{(i)}$ for some $1 \le i \le n$ is also labelled with index i. Each substring of S is uniquely represented by either an explicit or an implicit node of $\mathcal{T}(S)$, called its *locus*. Once $\mathcal{T}(S)$ is constructed, it can be traversed in a depth-first manner to compute the string-depth $\mathcal{D}(v)$ for each explicit node v. The suffix tree of a string of length n, over an integer alphabet, can be computed in time and space $\mathcal{O}(n)$ [41]. Each explicit node of the suffix tree is able to access its (explicit) parent. In the case of integer alphabets, in order to access the child of an explicit node by the first character of its edge label in $\mathcal{O}(1)$ time, perfect hashing [47] can be used.

A *generalized suffix tree* (GST) of strings $S_1, \dots, S_k$, denoted by $\mathcal{T}(S_1, \dots, S_k)$, is the suffix tree of $X = S_1\#_1 \dots S_k\#_k$, where $\#_1, \dots, \#_k$ are distinct end-markers.

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 $\mathsf{lcp}(S_{(r)}, S_{(s)})$. An $\mathcal{O}(n)$-sized lowest common ancestor data structure can be constructed over the suffix tree of S in $\mathcal{O}(n)$ time [22], 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* (textsflcs) queries in the same complexity. The lcp and textsflcs queries are also known as *longest common extension* (LCE) queries.

Suffix Array The *suffix array* of a string S of length n, denoted by $\mathsf{SA}(S)$, is an integer array of size $n+1$ storing the starting positions of all (lexicographically) sorted suffixes of S, i.e. for all $1 < r \le n+1$ we have $S[\mathsf{SA}(S)[r-1]\,..\,n] < S[\mathsf{SA}(S)[r]\,..\,n]$. Note that we explicitly add the empty suffix to the array. The suffix array $\mathsf{SA}(S)$ corresponds to a pre-order traversal of all terminal nodes of the suffix tree $\mathcal{T}(S)$. We define a *generalized suffix array* of $S_1, \dots, S_k$, denoted $\mathsf{SA}(S_1, \dots, S_k)$, as $\mathsf{SA}(S_1\#_1 \dots S_k\#_k)$.

3 Internal LCS Queries

In this section we consider LCS queries in the internal model. In the most general LCS queries, we are given strings S and T and upon query we are to report an LCS between a substring of S and a substring of T. We show a conditional lower bound for data structures for such queries in the next subsection. We then explore restricted versions of internal LCS queries and design efficient solutions for them. The developed data structures come handy in Sect. 4.

3.1 A Lower Bound Based on Set Disjointness

In the Set Disjointness problem, we are given a collection of m sets $S_1, S_2, \dots, S_m$ of total size N from some universe U for preprocessing in order to answer queries on the emptiness of the intersection of some two query sets from the collection. Goldstein et al. [52] demonstrated conditional hardness of Set Disjointness with regard to its space-query time tradeoff. Specifically, Goldstein et al. state the following conjecture.

Conjecture 1 (Strong Set Disjointness Conjecture) *Any data structure for the Set Disjointness problem that answers queries in t time must use space $\tilde{\Omega}(N^2/t^2)$.*[3]

Conjecture 1 is a generalization of the Set Disjointness conjecture stating that any data structure for the Set Disjointness problem with constant query time must use $\tilde{\Omega}(N^2)$ space [36, 69]. Unconditional lower bounds for the space-time tradeoff of the Set Disjointness were proven by Dietz et al. [40] and Afshani and Nielsen [4] for specific models of computation. Specifically, the results of [4] imply that Conjecture 1 is true in the pointer machine model.

Theorem 1 *Any data structure answering internal LCS queries for two strings, each of length at most n, in t time must use $\tilde{\Omega}(n^2/t^2)$ space, unless the Strong Set Disjointness Conjecture is false.*

[3] The $\tilde{\Omega}(\cdot)$ notation suppresses $\log^{\mathcal{O}(1)} n$ factors for inputs of size n.

Proof We reduce the Set Disjointness problem to that of answering internal LCS queries as follows. Given sets $S_1, \ldots, S_m$ of total cardinality N, we construct a string T of length $n = N$ that consists of the concatenation of the elements of the sets, so that each set S_i corresponds to the substring $T[a_i \mathinner{.\,.} b_i]$ of T and $\{T[a_i], T[a_i + 1], \ldots, T[b_i]\} = S_i$. Further consider a copy T' of T. Then, for any two sets S_i and S_j, $S_i \cap S_j$ is empty if and only if the length of an LCS of $T[a_i \mathinner{.\,.} b_i]$ and $T'[a_j \mathinner{.\,.} b_j]$ is 0. The statement follows. □

The proof of Theorem 1 mimics the proof of Amir et al. [14] for the hardness of so-called two-range-LCP queries. In the two-range-LCP problem, one is to preprocess a string so that queries of the following type can be answered: given two ranges I and J, return $\max_{i \in I, j \in J} \mathsf{lcp}(i,j)$. Amir et al. [14] presented a data structure achieving the tradeoff stated in the above lower bound. Now note that a general internal LCS query can be reduced via binary search to $\mathcal{O}(\log n)$ two-range-LCP queries as follows. The length of an LCS between $S[a_1 \mathinner{.\,.} b_1]$ and $T[a_2 \mathinner{.\,.} b_2]$ is at least m if and only if the two-range-LCP on the concatenation of S and T with intervals $[a_1 \mathinner{.\,.} b_1 - m + 1]$ and $[|S| + a_2 \mathinner{.\,.} |S| + b_2 - m + 1]$ is at least m. We summarize the above discussion in the following statement.

Proposition 1 *Given two strings of total length n and a parameter $t \le \sqrt{n}$, there is an $\tilde{\mathcal{O}}(n^2/t^2)$-size data structure that answers internal LCS queries in $\tilde{\mathcal{O}}(t)$ time.*

3.2 Auxiliary Data Structures Over the Suffix Tree

We first recall some auxiliary data structures.

Orthogonal Range Maximum Queries Let $\mathcal{P}$ be a collection of n points in a D-dimensional grid with integer weights and coordinates of magnitude $\mathcal{O}(n)$. In a D-dimensional orthogonal range maximum query $\mathsf{RMQ}_{\mathcal{P}}([a_1, b_1] \times \cdots \times [a_D, b_D])$, given a hyper-rectangle $[a_1, b_1] \times \cdots \times [a_D, b_D]$, we are to report the maximum weight of a point from $\mathcal{P}$ in the rectangle. We assume that the point that attains this maximum is also computed. The following result is known.

Lemma 1 ([5, 6, 23]) *Orthogonal range maximum queries over a set of n weighted points in D dimensions, where $D = \mathcal{O}(1)$, can be answered in $\tilde{\mathcal{O}}(1)$ time with a data structure of size $\tilde{\mathcal{O}}(n)$ that can be constructed in $\tilde{\mathcal{O}}(n)$ time. In particular, for $D = 2$ one can achieve $\mathcal{O}(\log n)$ query time, $\mathcal{O}(n \log n)$ space, and $\mathcal{O}(n \log^2 n)$ construction time.*

Data Structures for Trees We say that $\mathcal{T}$ is a *weighted tree* if it is a rooted tree with integer weights on nodes, denoted by $\mathcal{D}(v)$, such that the weight of the root is zero and $\mathcal{D}(u) < \mathcal{D}(v)$ if u is the parent of v. We say that a node v is a *weighted ancestor* of a node u at depth ℓ if v is the highest ancestor of u with weight of at least ℓ.

Lemma 2 ([13]) *After $\mathcal{O}(n)$-time preprocessing, weighted ancestor queries for nodes of a weighted tree $\mathcal{T}$ of size n can be answered in $\mathcal{O}(\log\log n)$ time per query.*

The following corollary applies Lemma 2 to the suffix tree.

Corollary 1 *The locus of any substring $S[i\,..\,j]$ in $\mathcal{T}(S)$ can be computed in $\mathcal{O}(\log\log n)$ time after $\mathcal{O}(n)$-time preprocessing.*

If $\mathcal{T}$ is a rooted tree, for each non-leaf node u of $\mathcal{T}$ the *heavy edge* (u, v) is an edge for which the subtree rooted at v has the maximal number of leaves (in case of several such subtrees, we fix one of them). A *heavy path* is a maximal path of heavy edges. The path from a node v in $\mathcal{T}$ to the root is composed of *prefix fragments* of heavy paths interleaved by single non-heavy (compact) edges. Here a prefix fragment of a path π is a path connecting the topmost node of π with any of its nodes. We denote this decomposition by $H(v, \mathcal{T})$. The following observation is known.

Lemma 3 ([71]) *For a rooted tree $\mathcal{T}$ of size n and a node v, $H(v, \mathcal{T})$ has size $\mathcal{O}(\log n)$ and can be computed in $\mathcal{O}(\log n)$ time after $\mathcal{O}(n)$-time preprocessing.*

In the heaviest induced ancestors (HIA) problem, introduced by Gagie et al. [50], we are given two weighted trees $\mathcal{T}_1$ and $\mathcal{T}_2$ with the same set of n leaves, numbered 1 through n, and are asked queries of the following form: given a node v_1 of $\mathcal{T}_1$ and a node v_2 of $\mathcal{T}_2$, return an ancestor u_1 of v_1 and an ancestor u_2 of v_2 that have a leaf descendant with the same label, say i (we say that the ancestors u_1 and u_2 are *induced* by the leaf i), and maximum total weight. We also consider *special* HIA queries in which we are to find the heaviest ancestor of v_{3-j} that is induced with v_j, for a specified $j \in \{1, 2\}$. Gagie et al. [50] provide several trade-offs for the space and query time of a data structure for answering HIA queries, some of which were recently improved by Abedin et al. [3]. All of them are based on heavy-path decompositions $H(v_1, \mathcal{T}_1)$, $H(v_2, \mathcal{T}_2)$. In the following lemma we use the variant of the data structure from Section 2.2 in [50], substituting the data structure used from [27] to the one from [6] to obtain a trade-off with a specified construction time. It can be readily verified that their technique answers special HIA queries within the same complexity.

Lemma 4 ([50]) *HIA queries and special HIA queries over two weighted trees $\mathcal{T}_1$ and $\mathcal{T}_2$ of total size $\mathcal{O}(n)$ can be answered in $\mathcal{O}(\log^2 n)$ time, using a data structure of size $\mathcal{O}(n\log^2 n)$ that can be constructed in $\mathcal{O}(n\log^2 n)$ time.*

Amir et al. [11] observed that the problem of computing an LCS after a single edit operation at position i can be decomposed into two queries out of which we choose the one with the maximal answer: an occurrence of an LCS either avoids i or it covers i. The former case can be precomputed. The latter can be reduced to HIA queries over suffix trees. It can be formalized by our Three Substrings LCS problem.

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

The lemma below was implicitly shown in [11] and [3].

Lemma 5 *A* Three Substrings LCS *query with $W = T$ can be answered in $\mathcal{O}(\log^2 n)$ time using a data structure of size $\mathcal{O}(n \log^2 n)$ that can be constructed in $\mathcal{O}(n \log^2 n)$ time.*

In Sect. 3.4 we show a solution to the general version of the Three Substrings LCS problem using a generalization of the HIA queries. First, in Sect. 3.3, we show efficient data structures for answering simpler types of internal LCS queries.

3.3 Internal Queries for Special Substrings

We show how to answer internal LCS queries for a prefix or suffix of S and a prefix or suffix of T and for a substring of S and the whole T.

In the solutions we use the formula:

$$|\mathsf{LCS}(S[a\mathinner{.\,.}b], T[c\mathinner{.\,.}d])| = \max_{\substack{i=a,\dots,b,\\ j=c,\dots,d}} \{\min\{\mathsf{lcp}(S_{(i)}, T_{(j)}),\ b-i+1,\ d-j+1\}\}. \tag{1}$$

We also apply the following observation to create range maximum queries data structures over points constructed from explicit nodes of the GST $\mathcal{T}(S, T)$.

Observation 1 *Let S and T be two strings of length n each. We have*

$$\{\mathsf{lcp}(S_{(i)}, T_{(j)}) \ :\ i, j = 1, \dots, n\} \subseteq \{\mathcal{D}(v) \ :\ v \text{ is explicit in } \mathcal{T}(S, T)\}.$$

Lemma 6 *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.*

Proof For a node v of $\mathcal{T}(S, T)$ and $U \in \{S, T\}$ we define:

$$minPref(v, U) = \min\{i \ :\ \mathcal{L}(v) \text{ is a prefix of } U_{(i)}\},$$
$$maxPref(v, U) = \max\{i \ :\ \mathcal{L}(v) \text{ is a prefix of } U_{(i)}\}.$$

We assume that $\min \emptyset = \infty$ and $\max \emptyset = -\infty$. These values can be computed for all explicit nodes of $\mathcal{T}(S, T)$ in $\mathcal{O}(n)$ time in a bottom-up traversal of the tree.

We only consider computing $\mathsf{LCS}(S_{(a)}, T_{(b)})$ and $\mathsf{LCS}(S^{(a)}, T_{(b)})$ as the remaining cases can be solved by considering the reversed strings.

In the first case formula (1) has an especially simple form:

$$|\mathsf{LCS}(S_{(a)}, T_{(b)})| = \max_{i \geq a, j \geq b} \mathsf{lcp}(S_{(i)}, T_{(j)})$$

which lets us use orthogonal range maximum queries to evaluate it. For each explicit node v of $\mathcal{T}(S, T)$ with descendants from both S and T we create a point (x, y) with weight $\mathcal{D}(v)$, where $x = \mathit{maxPref}(v, S)$ and $y = \mathit{maxPref}(v, T)$. By Observation 1, the sought LCS length is the maximum weight of a point in the rectangle $[a, n] \times [b, n]$. This lets us also recover the LCS itself. The complexity follows from Lemma 1.

In the second case the formula (1) becomes:

$$|\mathsf{LCS}(S^{(a)}, T_{(b)})| = \max_{i \leq a, j \geq b} \min(\mathsf{lcp}(S_{(i)}, T_{(j)}),\ a - i + 1).$$

The result is computed in one of two steps depending on which of the two terms produces the minimum. First let us consider the case that $\mathsf{lcp}(S_{(i)}, T_{(j)}) < a - i + 1$. For each explicit node v of $\mathcal{T}(S, T)$ with descendants from both S and T we create a point (x, y) with weight $\mathcal{D}(v)$, where $x = \mathit{minPref}(v, S) + \mathcal{D}(v) - 1$ and $y = \mathit{maxPref}(v, T)$. The answer r_1 is the maximum weight of a point in the rectangle $[1, a - 1] \times [b, n]$.

In the opposite case we can assume that the resulting internal LCS is a suffix of $S^{(a)}$ that does not occur earlier in S. For each explicit node v of $\mathcal{T}(S, T)$ we create a point (x, y) with weight x', where $x' = \mathit{minPref}(v, S)$, $x = x' + \mathcal{D}(v) - 1$, and $y = \mathit{maxPref}(v, T)$. Let i be the minimum weight of a point in the rectangle $[a, n] \times [b, n]$. If $i \leq a$, then we set $r_2 = a - i + 1$. Otherwise, we set $r_2 = -\infty$.

In both cases we use the 2d RMQ data structure of Lemma 1. In the end, we return $\max(r_1, r_2)$ and the corresponding LCS. □

The following lemma provides an efficient solution for the other special case of internal LCS.

Lemma 7 *Let S and T be two strings of length at most n. After $\mathcal{O}(n)$-time preprocessing, one can compute an LCS between T and any substring of S in $\mathcal{O}(\log n)$ time.*

Proof We define $B[i] = \max_{j=1,\dots,|T|} \{\mathsf{lcp}(S_{(i)}, T_{(j)})\}$. The following fact was shown in [11]. Here we give a proof for completeness.

Claim ([11]) *The values $B\,[i]$ for all $i = 1, \dots, |S|$ can be computed in $\mathcal{O}(n)$ time.*

Proof For every explicit node v of $\mathcal{T}(S, T)$ let us compute, as $\ell(v)$, the length of the longest common prefix of $\mathcal{L}(v)$ and any suffix of T. The values $\ell(v)$ are computed in a top-down manner. If v has as a descendant a leaf from T, then clearly $\ell(v) = \mathcal{D}(v)$. Otherwise, we set $\ell(v)$ to the value computed for v's parent. Finally, the values $B\,[i]$ can be read at the leaves of $\mathcal{T}(S, T)$. □

The formula (1) can be written as:

$$|\mathsf{LCS}(S[a\,..\,b],T)| = \max_{a \le k \le b}\{\min(B[k], b-k+1)\}.$$

The function $f(k) = b - k + 1$ is decreasing. We are thus interested in the smallest $k_0 \in [a,b]$ such that $B[k_0] \ge b - k_0 + 1$. If there is no such k_0, we set $k_0 = b + 1$. This lets us restate the previous formula as follows:

$$|\mathsf{LCS}(S[a\,..\,b],T)| = \max(\max_{a \le k < k_0}\{B[k]\},\, b - k_0 + 1).$$

Indeed, for $a \le k < k_0$ we know that $\min(B[k], b-k+1) = B[k]$, for $k = k_0$ we have $\min(B[k], b-k+1) = b - k_0 + 1$, and for $k_0 < k \le b$ we have $\min(B[k], b-k+1) \le b-k+1 \le b-k_0+1$.

The final formula for LCS length can be evaluated in $\mathcal{O}(1)$ time with a data structure for range maximum queries that can be constructed in linear time [22] on $B[1], \dots, B[n]$, provided that k_0 is known. This lets us also recover the LCS itself.

Computation of k_0 The condition for k_0 can be stated equivalently as $B[k_0] + k_0 \ge b + 1$. We create an auxiliary array $B'[i] = B[i] + i$. To find k_0, we need to find the smallest index $k \in [a,b]$ such that $B'[k] \ge b + 1$. We can do this in time $\mathcal{O}(\log n)$ by performing a binary search for k in the range $[a, b]$ of B' using $\mathcal{O}(n)$-time preprocessing and $\mathcal{O}(1)$-time range maximum queries [22]. $\square$

3.4 Three Substrings LCS Queries

We show how to answer the general Three Substrings LCS queries.

A solution to a special case of Three Substrings LCS queries with $W = T$ was already implicitly presented by Amir et al. in [11]. It is based on the *heaviest induced ancestors* (HIA) problem on trees 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 quite technical. It relies on the construction of multidimensional grids for pairs of heavy paths (in heavy-path decompositions [71]) 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.

We use *extended* HIA queries that we define for two weighted trees $\mathcal{T}_1$ and $\mathcal{T}_2$ with the same set of n leaves, numbered 1 through n, as follows: given $1 \le a \le b \le |T|$, a node v_1 of $\mathcal{T}_1$, and a node v_2 of $\mathcal{T}_2$, return an ancestor u_1 of v_1 and an ancestor u_2 of v_2 such that:

1. u_1 and u_2 are induced by some i;
2. $\mathcal{D}(u_j) = d_j$ for $j = 1, 2$;
3. $a \le i - d_1$ and $i + d_2 \le b + 1$;
4. $d_1 + d_2$ is maximal.

We also consider *special extended* HIA queries, in which the condition $u_1 = v_1$ or the condition $u_2 = v_2$ is imposed. Both extended and special extended HIA queries can be answered efficiently using multidimensional range maximum queries.

The motivation of the above definitions (extended and special extended HIA queries) becomes clearer in the proof of Lemma 9. Intuitively, the additional hardness is due to the fact that we use this type of queries to answer Three Substrings LCS for an *arbitrary substring* $W = T[a\,..\,b]$ instead of $W = T$. To this end, we have extended the HIA queries and present a data structure to answer them efficiently. The proposed data structure is based on a non-trivial combination of heavy-path decompositions and multidimensional range maximum data structures.

Lemma 8 *Extended HIA queries and special extended HIA queries over two weighted trees $\mathcal{T}_1$ and $\mathcal{T}_2$ of total size $\mathcal{O}(n)$ can be answered in $\tilde{\mathcal{O}}(1)$ time after $\tilde{\mathcal{O}}(n)$ -time and space preprocessing.*

Proof We defer answering special extended HIA queries until the end of the proof. Let us consider heavy paths in $\mathcal{T}_1$ and $\mathcal{T}_2$. Let us assign to each heavy path π in $\mathcal{T}_j$, for $j = 1, 2$, a unique integer identifier of magnitude $\mathcal{O}(n)$ denoted by $id(\pi)$. For a heavy path π and $i \in \{1, \dots, n\}$, by $d(\pi, i)$ we denote the depth of the lowest node of π that has leaf i in its subtree.

We will create four collections of weighted points $\mathcal{P}^I$, $\mathcal{P}^{II}$, $\mathcal{P}^{III}$, $\mathcal{P}^{IV}$ in 6d. Let $i \in \{1, \dots, n\}$. There are at most $\log n + 1$ heavy paths on the path from leaf number i to the root of each of $\mathcal{T}_1$ and $\mathcal{T}_2$. For each such *pair* of heavy paths, π_1 in $\mathcal{T}_1$ and π_2 in $\mathcal{T}_2$, we denote $d(\pi_j, i)$, for $j = 1, 2$, by d_j and insert the point:

- $(id(\pi_1), id(\pi_2), d_1, d_2, i - d_1, i + d_2)$ to $\mathcal{P}^I$ with weight $d_1 + d_2$;
- $(id(\pi_1), id(\pi_2), d_1, d_2, i, i + d_2)$ to $\mathcal{P}^{II}$ with weight d_2;
- $(id(\pi_1), id(\pi_2), d_1, d_2, i - d_1, i)$ to $\mathcal{P}^{III}$ with weight d_1;
- $(id(\pi_1), id(\pi_2), d_1, d_2, i, i)$ to $\mathcal{P}^{IV}$ with weight 0.

Thus each collection contains $\mathcal{O}(n \log^2 n)$ points. We perform preprocessing for range maximum queries on each of the collections by applying Lemma 1.

Assume that we are given an extended HIA query for v_1, v_2, a, and b (inspect Fig. 1 for an illustration). We consider all the prefix fragments of heavy paths in $H(v_1, \mathcal{T}_1)$ and $H(v_2, \mathcal{T}_2)$. For $j = 1, 2$, let π'_j be a prefix fragment of heavy path π_j in $H(v_j, \mathcal{T}_j)$, connecting node x_j with its descendant y_j. Now suppose that for some i, $d(\pi_j, i) > \mathcal{D}(y_j)$; we essentially want to reassign i to y_j which is the deepest ancestor of i in π'_j. To this end, we define intervals $I_j = [\mathcal{D}(x_j), \mathcal{D}(y_j) - 1]$ and $I_j^\infty = [\mathcal{D}(y_j), \infty)$.

For each of the $\mathcal{O}(\log^2 n)$ pairs of prefix fragments π'_1 and π'_2 in the decompositions of the root-to-v_1 and root-to-v_2 paths, respectively, we ask four range maximum queries, to obtain the following values:

1. $\mathsf{RMQ}_{\mathcal{P}^I}(id(\pi_1), id(\pi_2), I_1, I_2, [a, \infty), (-\infty, b + 1])$ that corresponds to finding a pair of induced nodes $u_1 \in \pi'_1 \setminus \{y_1\}$ and $u_2 \in \pi'_2 \setminus \{y_2\}$;

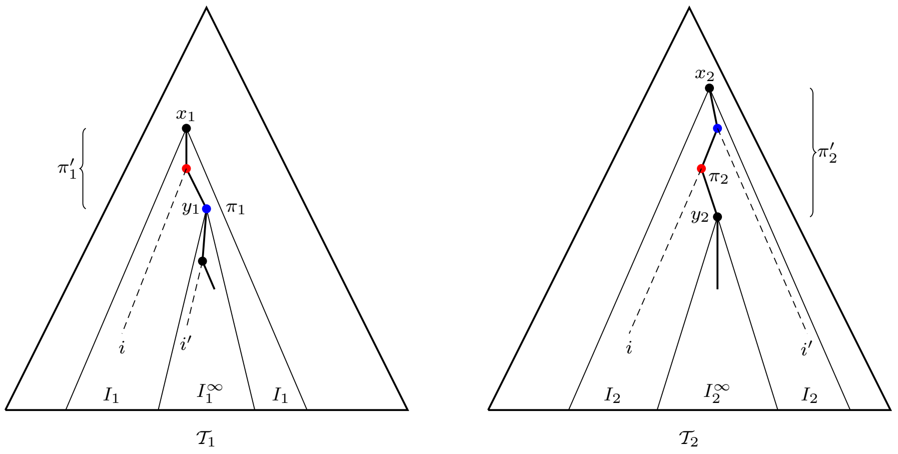

Fig. 1 Illustration of the notations used to handle a pair of prefix fragments π_1' and π_2'. The descendant leaves of x_j are implicitly partitioned at query time, by employing intervals I_j and I_j^∞, according to whether their deepest ancestor in π_j is a strict ancestor of y_j or not. For example, the pair of red nodes, induced by i, will be considered by the RMQ of type 1, while the pair of blue nodes, induced by i', by the RMQ of type 2, assuming that the last two constraints of the respective RMQ are satisfied (Color figure online)

2. $\mathcal{D}(y_1) + \mathsf{RMQ}_{\mathcal{P}^{II}}(id(\pi_1),\ id(\pi_2),\ I_1^\infty,\ I_2,\ [a+\mathcal{D}(y_1),\infty),\ (-\infty, b+1])$ that corresponds to finding a pair of induced nodes $u_1 = y_1$ and $u_2 \in \pi_2' \setminus \{y_2\}$;
3. $\mathcal{D}(y_2) + \mathsf{RMQ}_{\mathcal{P}^{III}}(id(\pi_1),\ id(\pi_2),\ I_1,\ I_2^\infty,\ [a,\infty),\ (-\infty, b+1-\mathcal{D}(y_2)])$ that corresponds to finding a pair of induced nodes $u_1 \in \pi_1' \setminus \{y_1\}$ and $u_2 = y_2$;
4. $\mathcal{D}(y_1) + \mathcal{D}(y_2) + \mathsf{RMQ}_{\mathcal{P}^{IV}}(id(\pi_1),\ id(\pi_2),\ I_1^\infty,\ I_2^\infty,\ [a+\mathcal{D}(y_1),\infty),\ (-\infty, b+1-\mathcal{D}(y_2)])$ that corresponds to checking if y_1 and y_2 are induced.

If an RMQ concerns an empty set of points, it is assumed to return $-\infty$. We return the point that yielded the maximal value.

Hence, an extended HIA query reduces to $\mathcal{O}(\log^2 n)$ range maximum queries in collections of points in 6d of size $\mathcal{O}(n \log^2 n)$. A special extended HIA query can be answered in a simpler way by asking just $\mathcal{O}(\log n)$ RMQs, where one of the prefix fragments π_1', π_2' reduces always to a single node. The statement follows. □

The proof of the following lemma is very similar to the proof of Lemma 5; we simply use extended HIA queries instead of the regular ones.

Lemma 9 *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.*

Proof We construct $\mathcal{T}_1 = \mathcal{T}(T^R)$, $\mathcal{T}_2 = \mathcal{T}(T)$, and the data structure for computing the loci of substrings (Corollary 1). The leaf corresponding to prefix $T^{(i-1)}$ in $\mathcal{T}_1$ and to suffix $T_{(i)}$ in $\mathcal{T}_2$ are labeled with i. Let $W = T[a\,..\,b]$ be an occurrence of W in T. If we treat $\mathcal{T}_1$ and $\mathcal{T}_2$ as weighted trees over the set of explicit nodes, then we have:

Claim *Let u_1 and u_2 be explicit nodes of $\mathcal{T}_1$ and $\mathcal{T}_2$, respectively, and $d_1 = \mathcal{D}(u_1)$, $d_2 = \mathcal{D}(u_2)$. Then $\mathcal{L}(u_1)^R\mathcal{L}(u_2)$ is a substring of W if and only if u_1 and u_2 are induced by i such that $a \le i - d_1$ and $i + d_2 - 1 \le b$.*

Proof ($\Rightarrow$) The string $\mathcal{L}(u_1)^R\mathcal{L}(u_2)$ is a substring of $W = T[a\,..\,b]$, so there exists an index $i \in [a\,..\,b]$ such that $\mathcal{L}(u_1)^R$ is a suffix of $T[a\,..\,i-1]$ and $\mathcal{L}(u_2)$ is a prefix of $T[i\,..\,b]$. This implies that u_1 and u_2 are induced by i and that:

$$d_1 = |\mathcal{L}(u_1)^R| \le |T[a\,..\,i-1]| = i - a \quad \text{and} \quad d_2 = |\mathcal{L}(u_2)| \le |T[i\,..\,b]| = b - i + 1.$$

($\Leftarrow$) If u_1 and u_2 are induced by i, then $\mathcal{L}(u_1)^R$ occurs as a suffix of $T^{(i-1)}$ and $\mathcal{L}(u_2)$ occurs as a prefix of $T_{(i)}$. By the inequalities $a \le i - d_1$ and $i + d_2 - 1 \le b$, $\mathcal{L}(u_1)^R\mathcal{L}(u_2)$ is a substring of $T[a\,..\,b] = W$. □

Assume we are given a Three Substrings LCS query for U, V, and $W = T[a\,..\,b]$. Let v_1 be the locus of U^R in $\mathcal{T}_1$ and v_2 be the locus of V in $\mathcal{T}_2$. By the claim, if both v_1 and v_2 are explicit, then the problem reduces to an extended HIA query for v_1 and v_2. Otherwise, we ask an extended HIA query for the lowest explicit ancestors of v_1 and v_2 and special extended HIA queries for the closest explicit descendant of v_j and v_{3-j} for $j \in \{1, 2\}$ such that v_j is implicit. □

4 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 2 LCS after One Substitution Per String *can be computed in $\tilde{\mathcal{O}}(1)$ time after $\tilde{\mathcal{O}}(n)$-time and space preprocessing.*

4.1 LCS Contains No Changed Position

It suffices to apply internal LCS queries of Lemma 6 four times: each time for one of $S^{(i-1)}, S_{(i+1)}$ and one of $T^{(j-1)}, T_{(j+1)}$.

4.2 LCS Contains a Changed Position in Exactly One of the Strings

We use the following lemma that encapsulates one of the main techniques of [11]. 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. [45].

Let S be a string of length n. Given a substring U of S, we denote by $range_S(U)$ the range in the $\mathsf{SA}(S)$ that represents the suffixes of S that have U as a prefix. Every node u in the suffix tree $\mathcal{T}(S)$ corresponds to an SA range $range_S(\mathcal{L}(u))$.

Lemma 10 *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.*

Proof Let X be a string of length n. We can precompute $range_X(\mathcal{L}(u))$ for all explicit nodes u in $\mathcal{T}(X)$ in $\mathcal{O}(n)$ time while performing a depth-first traversal of the tree. We use the following result by Fischer et al. [45].

Claim ([45]) *Let P and Q be two substrings of X and assume that $\mathsf{SA}(X)$ is known. Given $range_X(P)$ and $range_X(Q)$, $range_X(PQ)$ can be computed in time $\mathcal{O}(\log\log n)$ after $\mathcal{O}(n \log\log n)$-time and $\mathcal{O}(n)$-space preprocessing.*

We use the data structure of the claim for the generalized suffix array $\mathsf{SA}(S,T)$. The range of a substring P is denoted as $range_{S,T}(P)$. We assume that each element of $\mathsf{SA}(S,T)$ stores a 1 if and only if it originates from T and prefix sums of such values are stored. This lets us check if a given range of $\mathsf{SA}(S,T)$ contains any suffix of T in $\mathcal{O}(1)$ time. We also use the GST $\mathcal{T}(S,T)$.

By Corollary 1, the loci of P and Q in $\mathcal{T}(S,T)$ can be computed in $\mathcal{O}(\log\log n)$ time. This lets us recover the ranges $range_{S,T}(P)$ and $range_{S,T}(Q)$. By the claim, we can compute $range_{S,T}(PQ)$ in $\mathcal{O}(\log\log n)$ time. Then we check if PQ is a substring of T by checking if the resulting range contains a suffix of T; as already mentioned, this can be done in $\mathcal{O}(1)$ time. The data structures can be constructed in $\mathcal{O}(n \log\log n)$ time and use $\mathcal{O}(n)$ space. This concludes the proof of the first part of the lemma.

As for the second part, it suffices to apply binary search over P to find the longest prefix P' of P that is a substring of T. If $P' = P$, we apply binary search to find the longest prefix Q' of Q such that PQ' is a substring of T. Binary searches result in additional $\log n$-factor in the query complexity. The approach for computing the longest suffix is analogous. □

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 10(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 $S[1\,..\,i-1]$;
- V is the longest substring of T that is equal to a prefix of $\alpha S[i+1\,..\,|S|]$.

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 two Three substrings LCS queries: one with $W = T^{(j-1)}$ and one with $W = T_{(j+1)}$, for which we rely on Lemma 5.

4.3 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. [62] 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 11

(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) [62] *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.*

We next show an algorithm that finds a longest string $S'[i_\ell\,..\,i_r] = T'[j_\ell\,..\,j_r]$ such that $i_\ell \le i \le i_r$ and $j_\ell \le j \le j_r$ for the given indices i, j. Let us assume that $i - i_\ell \le j - j_\ell$; the symmetric case can be treated analogously. We have that $U \overset{\text{def}}{=} S'[i+1\,..\,i_\ell + j - j_\ell - 1] = T'[j_\ell + i - i_\ell + 1\,..\,j-1]$ as shown in Fig. 2. ($U = \varepsilon$ 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 11. 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:

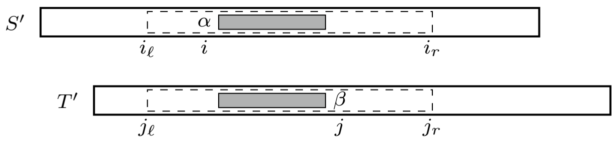

Fig. 2 Occurrences of an LCS of S' and T' containing both changed positions are denoted by dashed rectangles. Occurrences of U at which an LCS is aligned are denoted by gray rectangles

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

The setting is presented in Fig. 3. It can be readily verified (inspect Fig. 3) that a longest common substring aligned at the border of length $u - wp$, for $w \in [0, \ell - 1]$, is equal to

$$\begin{aligned} g(w) &= textsf{lcs}(X_2^R(P_2^R)^w, Y_2^R) + u - wp + \mathsf{lcp}(P_1^w X_1, Y_1) \\ &= \mathsf{lcp}(P_2^w X_2, Y_2) + \mathsf{lcp}(P_1^w X_1, Y_1) + u - wp. \end{aligned}$$

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 2 *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:

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

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

Example 2 Let $P = \mathtt{ab}$, $X = \mathtt{abbabba}$ and $Y = \mathtt{ababababbba}$. Further let $w = 2$. Then $f(2) = \mathsf{lcp}(P^2X, Y)$ can be computed as $b + \mathsf{lcp}(X[a+1\,..\,|X|], Y[b+1\,..\,|Y|]) = 7$ because $a + wp = b = 6$, where $p = |P| = 2$, $a = \mathsf{lcp}(P^\infty, X) = 2$ and $b = \mathsf{lcp}(P^\infty, X) = 6$. Indeed the longest common prefix of P^2X and Y is $\mathtt{abababb}$ and after that we have a mismatch: $X[4] = \mathtt{a} \neq Y[8] = \mathtt{b}$.

By Observation 2, $g(w)$ 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

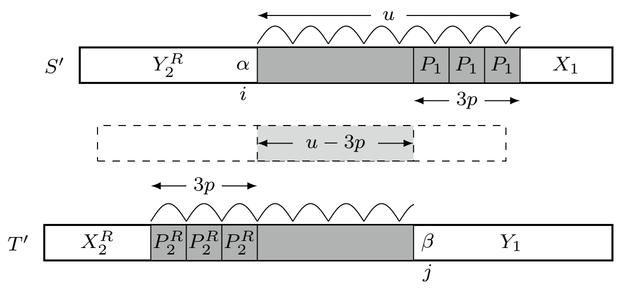

Fig. 3 A border of length u is denoted by dark gray rectangles. An LCS aligned at a border of length $u-3p$, which is in the same arithmetic sequence, is denoted by the dashed rectangle

$S'\#T'$, 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 Sects. 4.1–4.3, we arrive at Theorem 2.

5 Fully Dynamic 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. [51] for maintaining a string collection $\mathbf{W}$ under the following operations.

- makestring(W): insert a non-empty string W;
- concat(W_1, W_2): insert W_1W_2 to $\mathbf{W}$, for $W_1, W_2 \in \mathbf{W}$;
- split(W, i): split the string W at position i and insert both resulting strings to $\mathbf{W}$, for $W \in \mathbf{W}$;
- lcp (W_1, W_2): return the length of the longest common prefix of W_1 and W_2, for $W_1, W_2 \in \mathbf{W}$.

Lemma 12 (Gawrychowski et al. [51]) *A persistent collection $\mathbf{W}$ of strings of total length n can be dynamically maintained under operations* makestring(W), concat(W_1, W_2), split(W, i), *and* lcp (W_1, W_2) *with the operations requiring time $\mathcal{O}(\log n + |W|)$, $\mathcal{O}(\log n)$, $\mathcal{O}(\log n)$ all w.h.p. and $\mathcal{O}(1)$, respectively.*

Note that our results that make use of Lemma 12 hold w.h.p.

Lemma 13 *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 $m = \mathcal{O}(n)$* LCE *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.*

Proof Our preprocessing stage consists of a single makestring operation, requiring time $\mathcal{O}(n + \log n) = \mathcal{O}(n)$. Then each edit operation can be performed in $\mathcal{O}(\log n)$ time, using $\mathcal{O}(1)$ split, makestring(α), $\alpha \in \Sigma$, and concat operations. An LCE query can be answered by two split operations and a single lcp operation. The length of each of the strings on which split and concat operations are performed is $\mathcal{O}(n)$ and hence the cumulative length of the elements of the collection increases by $\mathcal{O}(n)$ with each performed operation. Since we perform $\mathcal{O}(n)$ operations in total, the cumulative length is $\mathcal{O}(n^2)$ and the time complexities of the statement follow by Lemma 12. □

Remark 1 A lemma similar to the above, based on a simple application of Karp-Rabin fingerprints [58], and avoiding the use of the heavy machinery underlying Lemma 12, can be proved.

Let us now proceed to the fully dynamic LCS problem. We first consider the case where the length of the sought LCS is bounded by some d; we call this problem d -Bounded-Length LCS.

Lemma 14 *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 $\mathbf{U}$ and $\mathbf{V}$ 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 binary search trees (balanced BSTs) $B_{\mathbf{X}}$, with respect to the lexicographical order, containing the elements of $\mathbf{X}$, for $\mathbf{X} = \mathbf{U}, \mathbf{V}$, stored as substrings. We can search in these balanced BSTs in $\mathcal{O}(\log^2 n)$ time since a comparison in it is an lcp query, which requires $\mathcal{O}(\log n)$ time by Lemma 13, possibly followed by a character comparison (which can be performed in $\mathcal{O}(\log n)$ time using the data structure of [51]). Each node of $B_{\mathbf{X}}$ will maintain a counter denoting its multiplicity in $\mathbf{X}$. Let $\mathbf{Y} = \mathbf{U} \cup \mathbf{V}$; we do not use $\mathbf{Y}$ in the algorithm, we just introduce it for conceptual convenience.

Observation 3 *The length of the LCS of length at most d is equal to the maximum lcp between pairs of consecutive substrings in (the sorted) $\mathbf{Y}$ that originate from different strings.*

During preprocessing, we compute the lcp's of all pairs described in Observation 3 and store them in a max heap H. To each element of the heap, we store a pointer from the nodes $u \in B_{\mathbf{U}}, v \in B_{\mathbf{V}}$ it originates from.

Each edit in S or T yields $\mathcal{O}(d)$ deletions and $\mathcal{O}(d)$ insertions of substrings in each of $\mathbf{U}$, $\mathbf{V}$ and $\mathbf{Y}$. 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) $\mathbf{Y}$, originating from different strings. We observe that upon the insertion/deletion of a string P, only pairs involving P, $pred_{\mathbf{U}}(P)$, $pred_{\mathbf{V}}(P)$, $succ_{\mathbf{U}}(P)$ and $succ_{\mathbf{V}}(P)$ 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 13, LCE queries (and heap updates) require $\mathcal{O}(\log n)$ time each. Finally, we return the maximum element of the heap. $\square$

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

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

Lemma 15 ([26, 65]) *For a positive integer d 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 setting is as follows (inspect also Fig. 4). 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 Fig. 4) and then merge these partial results accordingly.

For this we use the following auxiliary problem that was introduced in [28].

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

An efficient solution to this problem was shown in [28] (and, implicitly, in [38, 46]).

Lemma 16 ([28]) Two String Families LCP *can be solved in* $\mathcal{O}(|\mathbf{F}| + N \log N)$ *time, where* $N = |\mathbf{P}| + |\mathbf{Q}|$.

Theorem 3 Fully Dynamic LCS *on two strings, each of length up to n, can be solved in* $\tilde{\mathcal{O}}(n^{2/3})$ *time per edit operation w.h.p., using* $\tilde{\mathcal{O}}(n)$ *space, after* $\tilde{\mathcal{O}}(n)$*-time preprocessing.*

Proof Let us consider an integer $d \in [1, n]$. For lengths up to d, we use the algorithm for the d-Bounded-Length LCS problem of Lemma 14. 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 15).

We consider the following families of pairs of strings:

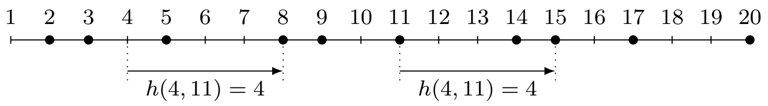

Fig. 4 An example of a 6-cover $\mathbf{S}_{20}(6) = \{2,3,5,8,9,11,14,15,17,20\}$, with the elements marked as black circles. For example, we may have $h(4,11) = 4$ since $4+4,\ 11+4 \in \mathbf{S}_{20}(6)$

$$\mathbf{P} = \{\ ((S[1\mathinner{.\,.}i-1])^R,\ S[i\mathinner{.\,.}|S|])\ :\ i \in A\ \},$$
$$\mathbf{Q} = \{\ ((T[1\mathinner{.\,.}i-1])^R,\ T[i\mathinner{.\,.}|T|])\ :\ i \in A\ \}.$$

We define $\mathbf{F}$ as the family of strings that occur in the pairs from $\mathbf{P}$ and $\mathbf{Q}$. Then maxPairLCP($\mathbf{P}, \mathbf{Q}$) equals the length of the sought LCS, provided that it is at least d.

Note that $|\mathbf{P}|, |\mathbf{Q}|, |\mathbf{F}|$ are $\mathcal{O}(n/\sqrt{d})$. A compact trie $\mathcal{T}(\mathbf{F})$ can be constructed in $\tilde{\mathcal{O}}(|\mathbf{F}| \log |\mathbf{F}|)$ time by sorting all the strings (using lcp-queries) and then adding them to the trie in lexicographic order; see [37]. We then use the solution to Two String Families LCP which takes $\tilde{\mathcal{O}}(n/\sqrt{d})$ time. We set $d = \lfloor n^{2/3} \rfloor$, rebuild the data structure of Lemma 13 that allows for efficient LCE queries after every $k = \lfloor n^{1/3} \rfloor$ edit operations and apply a standard timeslicing deamortization technique to obtain the stated complexities.[4] □

6 Fully Dynamic Longest Repeat

In the longest repeat problem we are given as input a string S of length n and we are to report a longest substring that occurs at least twice in S. This can be done in $\mathcal{O}(n)$ time and space [78]. In the fully dynamic longest repeat problem we are given as input a string S, which we are to maintain under subsequent edit operations, so that after each operation we can efficiently return a longest repeat in S. The application of our techniques for the LCS problem is quite straightforward, which is not surprising given the connection between the two problems. In what follows we briefly discuss the modifications required to the algorithms presented for the two subcases which we have decomposed the LCS problem into; we decompose the longest repeat problem in an analogous manner.

Lemma 14 covers the case that the LCS is short. It provides an efficient way to maintain all substrings of S and T of a specified length sorted lexicographically and maintaining the maximum value among the lcps of all pairs of consecutive substrings in this sorted list, originating from different strings. Here we can simply maintain this information for one string.

Lemma 16 finds long enough substrings that are common in both strings and are hence guaranteed to be anchored in a pair of positions in the difference cover.

[4] See Lemma 18 for a formalization of this deamortization technique in a similar setting.

Let A be this difference cover. Recall that we construct a compact trie $\mathcal{T}(\mathbf{F})$ of a family of strings $\mathbf{F}$, defined in terms of the input strings, the difference cover and the updated positions. Then, given $\mathcal{T}(\mathbf{F})$ and sets $\mathbf{P}, \mathbf{Q} \subseteq \mathbf{F}^2$ we efficiently compute $\max\{\mathsf{lcp}(P_1, Q_1) + \mathsf{lcp}(P_2, Q_2) : (P_1, P_2) \in \mathbf{P} \text{ and } (Q_1, Q_2) \in \mathbf{Q}\}$ by using an efficient solution to the Two String Families LCP problem. Sets $\mathbf{P}$ and $\mathbf{Q}$ essentially allow us to distinguish between substrings of S' and T' in $\mathbf{F}$. We adapt this for the longest repeat problem with an extra $\log n$ factor in the complexities. We build $\mathcal{O}(\log n)$ copies of the respective tree $\mathcal{T}(\mathbf{F})$, where in the j-th copy, $0 \le j \le \log n$, we set:

$$\mathbf{P} = \{\, ((S[1\,..\,i-1])^R,\, S[i\,..\,|S|]) \;:\; i \in A\,,\; \lfloor i/2^j \rfloor = 2m \,\},$$
$$\mathbf{Q} = \{\, ((S[1\,..\,i-1])^R,\, S[i\,..\,|S|]) \;:\; i \in A\,,\; \lfloor i/2^j \rfloor = 2m+1 \,\}$$

where m are integer.

Note that since $\mathbf{P}$ and $\mathbf{Q}$ are disjoint, in none of the tries do we align a fragment of S with itself. It is also easy to see that any two positions in A will be in different sets at one of the copies of $\mathcal{T}(\mathbf{F})$.

Theorem 4 *A longest repeat of a string of length up to n can be maintained in $\tilde{\mathcal{O}}(n^{2/3})$ time w.h.p. per edit operation, using $\tilde{\mathcal{O}}(n)$ space, after $\tilde{\mathcal{O}}(n)$ -time preprocessing.*

7 General Scheme for Dynamic Problems on Strings

We now present a general scheme for dynamic problems on strings. Let the input be a string S of length n. 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 $\mathcal{O}(t_n)$ ($s_n \ge n$ and $q_n(k) \ge k$ 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 (possibly empty) 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 17 *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 $S' = F_1 \dots F_k$ 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 $F_1 \dots F_k$ of S. For problems in which we aim at computing the longest substring of S that satisfies a certain property there are two cases.

- *Case 1*: the sought substring occurs inside one of the substrings F_i. This requires us to compute the solution to a certain internal pattern matching problem.
- *Case 2*: it contains the boundary between some two substrings F_i and F_{i+1}. We call Case 2 *cross-substring* queries. Note that certain internal queries may arise in cross-substring queries as well.

Finally, let us formalize the time slicing deamortization technique.

Lemma 18 *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 $q_n(k) \geq k$ 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 $\mathcal{O}(q_n(\kappa))$, where κ is such that $q_n(\kappa) = (t_n + n)/\kappa$, using $\mathcal{O}(s_n)$ space.*

Proof We first build $\mathcal{D}$ for the input string. The k-substring of S after the subsequent edit operations is stored. We keep a counter C of the number of queries answered since the point in time to which our data structure refers; if $C \leq 2\kappa$ and a new edit operation occurs, we create the $(2C+1)$-substring representing the string from the $(2C-1)$-substring we have using Lemma 17 in time $\mathcal{O}(C) = \mathcal{O}(\kappa)$ and answer the query in time $\mathcal{O}(q_n(C)) = \mathcal{O}(q_n(\kappa))$.

For convenience let us assume that κ is an integer. As soon as $C = \kappa$, we start recomputing the data structure $\mathcal{D}$ for the string after all the edit operations so far, but we allocate this computation so that it happens while answering the next κ queries. First we create a working copy of the string in $\mathcal{O}(n)$ time and then construct the data structure in $\mathcal{O}(t_n)$ time.

When $C = 2\kappa$, we set C to κ, dispose of the original data structure and string and start using the new ones for the next κ queries while computing the next one.

The following invariant is kept throughout the execution of the algorithm: the data structure being used refers to the string(s) at most 2κ edit operations before.

Hence, the query time is $\mathcal{O}(q_n(\kappa))$. The extra time spent during each query for computing the data structure is also $\mathcal{O}((t_n + n)/\kappa) = \mathcal{O}(q_n(\kappa))$ since the $\mathcal{O}(t_n + n)$ time is split equally among κ queries. At every point in the algorithm we store at most two copies of the data structure. □

8 Fully Dynamic Longest Palindrome Substring

Palindromes (also known as symmetric strings) are one of the fundamental concepts on strings with applications in computational biology (see, e.g., [54]). A recent progress in this area was the design of an $\mathcal{O}(n \log n)$-time algorithm for partitioning a string into the minimum number of palindromes [44, 57] (that was improved to $\mathcal{O}(n)$ time [24] afterwards). The main combinatorial insight of these results is that the set of lengths of suffix palindromes of a string can be represented as a logarithmic number of arithmetic progressions, each of which consists of palindromes with the same shortest period. Funakoshi et al. [48] use this fact to present a data structure for computing a longest palindrome substring of a string after a single edit operation. This problem is called Longest Palindrome Substring after One Edit. They obtain $\mathcal{O}(\log \log n)$-time queries with a data structure of $\mathcal{O}(n)$ size that can be constructed in $\mathcal{O}(n)$ time. We present a fully dynamic algorithm with $\tilde{\mathcal{O}}(\sqrt{n})$-time queries for this problem.

A *palindrome* is a string U such that $U^R = U$. For a string S, by $LSPal(S)$ let us denote a longest substring of S that is a palindrome. We first show a data structure with $\tilde{\mathcal{O}}(n)$ preprocessing time and $\tilde{\mathcal{O}}(1)$ time for internal longest palindrome substring queries; it is based on orthogonal range maximum queries. We then show a solution to the following auxiliary problem.

k-Substring LSPal
Input: A string S of length n
Query: $LSPal(S')$ where $S' = F_1 \dots F_k$ is a k-substring of S

The *center* of a palindrome substring $S[i\,..\,j]$ is $\frac{i+j}{2}$. A palindrome substring $S[i\,..\,j]$ of S is called *maximal* if $i = 1$, or $j = n$, or $S[i-1\,..\,j+1]$ is not a palindrome. For short, we call maximal palindrome substrings MPSs. By $\mathbf{M}(S)$ we denote the set of all MPSs of S. For each integer or half-integer center between 1 and n there is exactly one MPS with this center, so $|\mathbf{M}(S)| = 2n - 1$. The set $\mathbf{M}(S)$ can be computed in $\mathcal{O}(n)$ time using Manacher's algorithm [66] or LCE queries [54].

8.1 Internal Queries

In an internal *LSPal* query we are to compute the longest palindrome substring of a given substring of S. In the following lemma we show that such queries can be reduced to 2d range maximum queries.

Lemma 19 *Let S be a string of length n. After $\mathcal{O}(n\log^2 n)$-time and $\mathcal{O}(n\log n)$-space preprocessing, one can compute the LSPal of a substring of S and the longest prefix/suffix palindrome of a substring of S in $\mathcal{O}(\log n)$ time.*

Proof *LSPal*($S[i\,..\,j]$) is a substring of an MPS of S with the same center. We consider two cases depending on whether the *LSPal* is equal to the MPS.

If this is the case, the MPS is a substring of $S[i\,..\,j]$. We create a 2d grid and for each $S[a\,..\,b] \in \mathbf{M}(S)$ we create a point (a, b) with weight $b-a+1$. The sought MPS can be found by an RMQ for the rectangle $[i,\infty)\times(-\infty, j]$.

In the opposite case, *LSPal*($S[i\,..\,j]$) is a prefix or a suffix of $S[i\,..\,j]$. We consider the case that it is a prefix of $S[i\,..\,j]$; the other case is symmetric. The longest prefix palindrome of $S[i\,..\,j]$ can be obtained by trimming to the interval of positions $[i\,..\,j]$, the MPS with the rightmost center, among the ones with starting position smaller than i and center at most $(i+j)/2$. To this end, we create a new 2d grid. For each $S[a\,..\,b] \in \mathbf{M}(S)$, we create a point $(a, a+b)$ with weight $a+b$. The answer to an RMQ on the rectangle $[i,\infty)\times(-\infty, i+j]$ is twice the center of the desired MPS of S.

In either case, we use Lemma 1 to answer a 2d RMQ; the complexity follows. □

If the answer to k -Substring *LSPal* contains none of the changed positions, it can be found by asking $k+1$ internal *LSPal* queries.

8.2 Cross-Substring Queries

Let us assume that the boundary between F_{i-1} and F_i is the closest one to the center of *LSPal*. In what follows, we consider the case that it lies to the left of the center. Then, the palindrome cut to the positions in F_i is a prefix palindrome of F_i. The opposite case is symmetric. In total, this gives rise to $2k$ cases that need to be checked.

The structure of palindromes being prefixes of a string has been well studied. It is known that a palindrome being a prefix of another palindrome is also its border, and a border of a palindrome is always a palindrome; see [44]. Hence, the palindromes being prefixes of F_i are the borders of the longest such palindrome, further denoted by U_0. We are interested in the borders of U_0.

The palindrome U_0 can be computed by Lemma 19. By Lemma 11, the set of border lengths of U_0 can be divided into $\mathcal{O}(\log n)$ arithmetic sequences. Each such sequence will be treated separately. Assume first that it contains at most two elements. Let f_i denote the starting position of F_i in S', for $i=1,\dots,k$. Then for each element u representing a palindrome U, the longest palindrome having the same center as U in S' has the length

$$u + 2\cdot \mathsf{lcp}(S'_{(f_i+u)}, ((S')^{(f_i-1)})^R).$$

This LCE query can be answered in $\mathcal{O}(\log n)$ time using Lemma 13 for $S\#S^R$.

Now assume that the arithmetic sequence has more than two elements. Let p be the difference of the arithmetic sequence, ℓ be its length, u be its maximum element, and

$$X = S'_{(f_i+u)}, \quad Y = ((S')^{(f_i-1)})^R, \quad P = S'[f_i + u - p + 2 \,..\, f_i + u - 1].$$

Then the longest palindrome having the same center as an element of this sequence has length

$$2 \cdot \max_{w=0}^{\ell-1} \{ \, \mathsf{lcp}(P^w X, Y) + \frac{1}{2}(u - wp) \, \}.$$

By Observation 2, this formula can be evaluated in $\mathcal{O}(\log n)$ time.

Over all arithmetic sequences, we obtain $\mathcal{O}(\log^2 n)$ query time.

8.3 Round-Up

The results of the two subsections can be combined into an algorithm for k -Substring LSPal.

Lemma 20 k -Substring LSPal *queries can be answered in* $\mathcal{O}(k \log^2 n)$ *time after* $\mathcal{O}(n \log^2 n)$*-time and* $\mathcal{O}(n \log n)$*-space preprocessing.*

Using the general scheme (Lemma 18), we obtain a solution to fully dynamic longest palindrome substring problem.

Theorem 5 *A longest palindrome substring of a string of length up to n can be maintained in* $\mathcal{O}(\sqrt{n} \log^2 n)$ *time per edit operation w.h.p., using* $\mathcal{O}(n \log n)$ *space, after* $\mathcal{O}(n \log^2 n)$*-time preprocessing.*

9 Fully Dynamic Longest Lyndon Substring

A *Lyndon string* is a string that is lexicographically smaller than all its suffixes [64]. (Let us recall that string S is smaller in the lexicographic order than string T, written as $S < T$, if S is a proper prefix of T or $S^{(i)} = T^{(i)}$ and $S[i+1] < T[i+1]$.) For example, `aabab` and `a` are Lyndon strings, whereas `abaab` and `abab` are not. Lyndon strings are an object of interest in combinatorics on words, especially due to the Lyndon factorization theorem [34] that asserts that every string can be uniquely decomposed into a non-decreasing sequence of Lyndon strings. More formally, the Lyndon factorization of a string S, denoted as LF_S, is the unique way of writing S as $L_1 \dots L_p$ where $L_1, \dots, L_p$ are Lyndon strings (called *factors*) that satisfy $L_1 \geq L_2 \geq \dots \geq L_p$ (if S is a Lyndon string, then its Lyndon factorization is composed of S itself). Recently, Lyndon strings have found important applications in algorithm design [68] and were used to settle a known conjecture on the number of maximal repetitions in a string [19, 20].

Urabe et al. [77] presented a data structure for computing a longest substring of a string being a Lyndon string in the restricted dynamic setting of a single edit that is reverted afterwards. This problem is called Longest Lyndon Substring after One Edit. Their data structure can be constructed in $\mathcal{O}(n)$ time and space and answers queries in $\mathcal{O}\mathcal{O}(\log n)$ time. A simple observation of [77] is the following.

Lemma 21 (Lemma 3 in [77]) *The longest Lyndon substring of a string is the longest factor in its Lyndon factorization.*

Thus, this work indirectly poses the question of maintaining the Lyndon factorization of a dynamic string. We first show how to answer queries for the Lyndon factorization of a given substring. We then present an algorithm that maintains a representation of the Lyndon factorization of a string with $\tilde{\mathcal{O}}(\sqrt{n})$-time queries in the fully dynamic setting.

9.1 Internal Queries

We consider the following internal Lyndon factorization queries in a static string:

- `longest`(i,j): computing the longest factor in $LF_{S[i\,.\,.\,j]}$;
- `count`(i,j): computing the number of factors in $LF_{S[i\,.\,.\,j]}$;
- `select`(i,j,t): computing the t-th factor in $LF_{S[i\,.\,.\,j]}$.

In the rest of this subsection, we show the following lemma.

Lemma 22 *Let S be a string of length n. After $\mathcal{O}(n)$-time preprocessing, given a substring $S[i\,.\,.\,j]$ of S, we can answer* `longest`(i,j), `count`(i,j) *and* `select`(i,j,t) *queries in time $\mathcal{O}(\log^2 n)$.*

Main Idea Urabe et al. [77] show how to efficiently compute a representation of a Lyndon factorization of a prefix of a string and of a suffix of a string. For the prefixes, their solution is based on the Lyndon representations of prefixes of a Lyndon string, whereas for the suffixes, they rely on the structure of the Lyndon tree (due to [21]). We combine the two approaches to obtain Lemma 22.

Let us start with the definition of a Lyndon tree of a Lyndon string W [21]. If W is not a Lyndon string, the tree is constructed for $\$W$, where $\$$ is a special character that is smaller than all the characters in W. An example of a Lyndon tree can be found in Fig. 5 below.

Definition 1 The *standard factorization* of a Lyndon string S is (U, V) if $S = UV$ and V is the longest proper suffix of S that is a Lyndon string. In this case, both U and V are Lyndon strings. The *Lyndon tree* of a Lyndon string S is the full binary

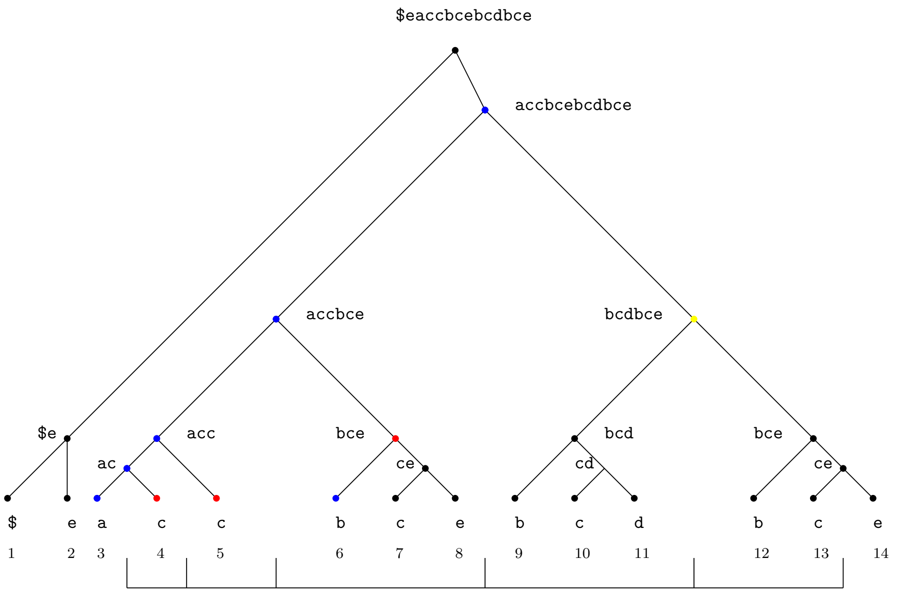

Fig. 5 The Lyndon tree of string $S =$ `$eaccbcebcdbce`

tree defined by recursive standard factorization of S. More precisely, S is the root; if $|S| > 1$, its left child is the root of the Lyndon tree of U, and its right child is the root of the Lyndon tree of V.

If v is a node of a binary tree, u is a strict ancestor of v (i.e. $u \neq v$), and w is the right child of u that is not an ancestor of v, then we call w a *right uncle* of v. By $U(v)$ we denote the list of all right uncles of v in bottom-up order. The following lemma was shown in [77].

Lemma 23 (Lemma 12 in [77]) *$LF_{S_{(j)}} = U(v)$, where v is the leaf of $LTree(S)$ that corresponds to $S[j-1]$.*

We say that S is a *pre-Lyndon string* if it is a prefix of a Lyndon string. The following lemma is a well-known property of Lyndon strings; see Lemma 10 in [77] or the book [59].

Lemma 24 ([59, 77]) *If S is a pre-Lyndon string, then there exists a unique Lyndon string X such that $S = X^kX'$ where X' is a proper prefix of X* and $k \geq 1$. *Moreover, $LF_S = \underbrace{X, X, \ldots, X}_{k \text{ times}}, LF_{X'}$.*

In this case LF_S can be represented in a *compact* form by simply writing the first part of the representation as $(X)^k$. Note that X' is a prefix of S and hence is also a pre-Lyndon string. Finally, the string X from Lemma 24 also satisfies the following property.

Observation 4 $|X|$ is the shortest period of S.

Proof Clearly $|X|$ is a period of S. If it was not the shortest period, then X would have a non-trivial period, hence a proper non-empty border B. Then, since $B < X$, X would not be a Lyndon string, a contradiction. □

The above properties are sufficient to determine Lyndon factorizations of prefixes and suffixes of a string S according to [77] as follows:

- To compute $LF_{S^{(j)}}$, take the minimal number of prefix factors in LF_S that cover $S^{(j)}$, trim the last of these factors accordingly, compute the Lyndon factorization of the trimmed factor by repeatedly using Lemma 24, and append it to the previous factors.
- To compute $LF_{S_{(j)}}$, take the list of right uncles of the leaf $S[j-1]$ as shown in Lemma 23.

We are now ready to describe $LF_{S[i\,..\,j]}$. Let v_1 and v_2 be the leaves of $LTree(S)$ that correspond to $S[i-1]$ and $S\,[j]$, respectively, w be their lowest common ancestor (LCA), and u be its right child. Then $LF_{S[i\,..\,j]}$ is determined by taking the list of right uncles $U(v_1)$ up to u, trimming the factor in u up to position j, and computing the Lyndon factorization of the trimmed factor according to Lemma 24.

Example 3 We consider the Lyndon string $S = \$$`eaccbcebcdbce`, whose Lyndon tree is presented in Fig. 5. Let us suppose that we want to compute the Lyndon factorization of the substring $S[4\,..\,13]$ = `ccbcebcdbc`. We first find the LCA of the leaves representing $S\,[3]$ and $S\,[13]$. The path from this LCA to the leaf representing $S\,[3]$ is shown in blue. The Lyndon factorization of $S[4\,..\,14]$ can be obtained by the right uncles of $S\,[3]$ (red and yellow nodes). In order to have $LF_{S[4\,..\,13]}$, the last right uncle (in yellow) needs to be trimmed. By Observation 4, we take the shortest period of the trimmed string $S[9\,..\,13]$ = `bcdbc` and obtain the Lyndon factorization `bcd`, `bc`. Thus we obtain the Lyndon factorization of $S[4\,..\,13]$, also depicted in Fig. 5, which is `c`, `c`, `bce`, `bcd`, `bc`.

In order to make this computation efficient we make use of the heavy path decomposition of $LTree(S)$. Each non-leaf node v stores, as $rc(v)$, the length of its right child provided that it is *not* present on its heavy path, or -1 otherwise. A balanced BST with all the nodes in the heavy path with positive values of rc given in bottom-up order is stored for every heavy path. It can be augmented in a standard manner to support the following types of queries on any subpath of the heavy path in $\mathcal{O}(\log n)$ time:

- longest-subpath: the maximal value rc on the subpath;

- count-subpath: the number of nodes with a positive value of rc on the subpath;
- select-subpath(i): the i-th node with a positive value of rc on the subpath.

These precomputations take $\mathcal{O}(n)$ time and space. Finally, a lowest common ancestor data structure can be constructed over the Lyndon tree in $\mathcal{O}(n)$ time and $\mathcal{O}(n)$ space [22] supporting LCA queries in $\mathcal{O}(1)$ time per query.

We represent $LF_{S[i..j]}$ as follows. First, the pair of nodes (v_1, w') where w' is the left child of w is stored. Second, the Lyndon factorization of a prefix u' of the right child u of w, which we compute by recursively applying Lemma 24. We can store the $LF_{u'}$ in a compact form in $\mathcal{O}(\log n)$ space. This representation can be computed in $\mathcal{O}(\log^2 n)$ time as follows:

- The LCA w of v_1 and v_2 is computed in $\mathcal{O}(1)$ time. Then w' is the left child of w.
- Each step of the recursive factorization of the pre-Lyndon string u' can be performed in $\mathcal{O}(\log n)$ time by employing the data structure of Kociumaka et al. [62] that can be constructed in $\mathcal{O}(n)$ time and answers internal Period Queries in $\mathcal{O}(\log n)$ time, due to Lemma 24 and Observation 4. The total number of steps is $\mathcal{O}(\log n)$ since each step at least halves the length of the string to be factorized.

Finally let us check that we can support the desired types of queries in $\mathcal{O}(\log^2 n)$ time:

- longest(i, j): We divide the path from v_1 to w' into $\mathcal{O}(\log n)$ subpaths of heavy paths and for each of them ask a longest-subpath query. This takes $\mathcal{O}(\log^2 n)$ time. We compare the maximum of the results with the maximum length of a factor in the second part of the representation, in $\mathcal{O}(\log n)$ time.
- count(i, j) is implemented analogously using count-subpath.
- select(i, j, t): We use count-subpath queries to locate the subpath that contains the t-th factor in the whole factorization or check that none of the subpaths contains it. In the first case, we use a select-subpath query. In the second case, we locate the correct factor in the second part of the representation in $\mathcal{O}(\log n)$ time.

We have thus proved Lemma 22.

9.2 Cross-Substring Queries

In this section we show how to compute a representation of the Lyndon factorization of a k-substring S' from the Lyndon factorizations of the k involved substrings. We want to be able to answer longest, count and select(t) queries for S'—not for substrings of S'. We rely on the following characterization of Lyndon factorizations of concatenations of strings.

Lemma 25 (see Lemmas 6 and 7 in [77]; originally due to [15, 39, 56]) *Assume that* $LF_U = (L_1)^{p_1}, \ldots, (L_m)^{p_m}$ *and* $LF_V = (L'_1)^{p'_1}, \ldots, (L'_{m'})^{p'_{m'}}$. *Then:*

(a) $LF_{UV} = (L_1)^{p_1}, \dots, (L_c)^{p_c}, Z^r, (L'_{c'})^{p'_{c'}}, \dots, (L'_{m'})^{p'_{m'}}$ *for some* $1 \le c \le m, 1 \le c' \le m'$, string Z, *and positive integer* r.

(b) *If* LF_U *and* LF_V *have been computed, then* c, c', Z, *and* r *from point (a) can be computed by* $\mathcal{O}(\log |LF_U| + \log |LF_V|)$ *lexicographic comparisons of strings, each of which is a 2-substring of UV composed of concatenations of a number of consecutive factors in* LF_U *or* LF_V.

We will represent $LF_{S'}$ for a k-substring S' of S as a sequence of elements of two types: powers of Lyndon substrings of S' and pairs of nodes (v, w) in $LTree(S)$ that denote the bottom-up list of right uncles of nodes on the path from v to w in the tree. To compute a representation of $LF_{S'}$ for $S' = F_1 \dots F_k$, we compute the representation of LF_{F_i} for all $i = 1, \dots, k$ defined in the above subsection and then repetitively apply Lemma 25. (Note that the representation of LF_{F_i} is of the desired form for all i.) Let $S'' = F_1 \dots F_{i-1}$ and assume that the desired representation of $LF_{S''}$ has been already computed. Then, by Lemma 25, a representation of $LF_{S''F_i}$ can be obtained by removing a number of trailing elements in $LF_{S''}$, a number of leading elements of LF_{F_i}, and merging them with at most one power of a Lyndon substring Z of S' in between. The removal of an element may be captured by (a) the removal of a power of a Lyndon substring, (b) the removal of a pair of nodes, (c) the alteration of some pair of nodes, corresponding to trimming the underlying path.

The length of our representation of $LF_{S'}$ is $\mathcal{O}(k \log n)$ and all its elements are stored in a left-to-right order in a balanced BST. For each element, the maximum length of a factor and the number of factors are also stored in the BST. The BST is augmented with the counts of Lyndon factors so that one can identify in logarithmic time the element of the representation that contains the t-th Lyndon factor in the whole factorization. This lets us implement the operation $\mathsf{select}(t)$ in $\mathcal{O}(\log(k \log n) + \log^2 n) = \mathcal{O}(\log^2 n)$ time (for $k \le n$) by first identifying the correct element of the representation and then selecting the appropriate Lyndon factor in this element. The $\mathsf{longest}$ and count operations are performed in $\mathcal{O}(1)$ time if their results are stored together with the representation.

Complexity Overall, the internal queries require $\mathcal{O}(k \log^2 n)$ time after $\mathcal{O}(n)$ -time and space preprocessing. Every application of Lemma 25 requires $\mathcal{O}(\log(k \log n)) = \mathcal{O}(\log k + \log \log n)$ lexicographic comparisons of 2-substrings of S', which gives $\mathcal{O}(\log k)$ if k is polynomial in n. The 2-substrings can be identified in $\mathcal{O}(\log^2 n)$ time by a select query and then compared using the data structure of Lemma 13. This lemma requires $\mathcal{O}(n)$ space and answers m LCE queries on a k-substring in $\mathcal{O}((k + m) \log n)$ time, which gives $\mathcal{O}(k \log k \log n) = \mathcal{O}(k \log^2 n)$ time over all applications of Lemma 25. Updating the maximum length and the count of Lyndon factors represented by the at most two trimmed paths per application of Lemma 25 requires $\mathcal{O}(\log^2 n)$ time. In total, $\mathcal{O}(k \log^3 n)$ time is required to compute a representation of $LF_{S'}$.

We have just proved the following lemma.

Lemma 26 *After an* $\mathcal{O}(n)$*-time and space preprocessing of a string* S *of length* n*, given a k-substring of* S*, for* $k \le n$*, we can process it in* $\mathcal{O}(k \log^3 n)$ *time, answering* longest *and* count *queries. We can then answer* select(t) *queries in* $\mathcal{O}(\log^2 n)$ *time each.*

Remark 2 One could define the Lyndon factorization of a string S as a unique way of expressing S as $(L_1)^{a_1} \dots (L_p)^{a_p}$ where $a_1, \dots, a_p \geq 1$ are integers and $L_1 > L_2 > \dots > L_p$. With this definition, the same complexities of operations as in Lemma 26 can be achieved for count queries, returning the length p of the representation, and select(t) queries for $1 \leq t \leq p$.

Let us recall that $LF_{S'}$ is stored using powers of Lyndon substrings of S and lists of right uncles on heavy paths of Lyndon trees as building blocks. Then the main modification to our approach is to store, for each right uncle, a bit stating if is is different from the previous right uncle on its heavy path. This additional data can be computed in constant time per right uncle using LCE queries. Then the count-subpath and select-subpath queries consider only the uncles for which this bit is set (treating the first right uncle on each heavy path separately).

9.3 Round-Up

We apply the time slicing technique (Lemma 18) to Lemma 26 in order to obtain a solution in the fully dynamic setting.

Theorem 6 *We can preprocess a string S of length n in $\mathcal{O}(n)$ time, and then process each edit operation in $\mathcal{O}(\sqrt{n}\log^{1.5} n)$ time w.h.p., returning the longest Lyndon substring and the size of the Lyndon factorization. We can then answer* select(t) *queries in* $\mathcal{O}(\log^2 n)$ time each.

10 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.

We conclude by summarizing the main results of a very recent work [29] on the dynamic LCS problem, which improves over some of our results. It was published after the submission of this manuscript. This work shows that the fully dynamic LCS problem admits a solution with $\tilde{\mathcal{O}}(1)$ update time, that uses $\tilde{\mathcal{O}}(n)$ space. The authors complement this upper bound by an unconditional lower bound: the update time of any polynomial-sized data structure for the fully dynamic LCS is $\Omega(\log n/\log\log n)$. They also consider the partially dynamic LCS problem, in which only one of the two strings is subject to updates. This problem has been considered in an earlier version of our work. For this problem, the authors show an $\tilde{\mathcal{O}}(n)$-space, $\mathcal{O}(\log^2 n)$-update time solution and an $\tilde{\mathcal{O}}(n^{1+\epsilon})$-space, $\mathcal{O}(\log\log n)$-update time solution, for any constant $\epsilon > 0$. On the lower bounds' side, they show that the update time of any $\tilde{\mathcal{O}}(n)$ -sized data structure for this problem is $\Omega(\log n/\log\log n)$.

Acknowledgements Amihood Amir was partially supported by Israel Science Foundation (ISF) Grant 1475/18, and Binational Science Foundation (BSF) Grant 2018141. Panagiotis Charalampopoulos was partially supported by a King's College London NMS Faculty Studentship, an Educational Grant by the A. G. Leventis Foundation and Israel Science Foundation (ISF) Grant 794/13. Jakub Radoszewski was supported by the "Algorithms for text processing with errors and uncertainties" project carried out within the HOMING programme of the Foundation for Polish Science co-financed by the European Union under the European Regional Development Fund, Project No. POIR.04.04.00-00-24BA/16.

References

1. Abboud, A., Backurs, A., Williams, V.V.: Tight hardness results for LCS and other sequence similarity measures. In: IEEE 56th Annual Symposium on Foundations of Computer Science, FOCS 2015, pp. 59–78. IEEE Computer Society (2015)
2. Abboud, A., Williams, R.R., Yu, H.: More applications of the polynomial method to algorithm design. In: Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015, San Diego, CA, USA, January 4–6, 2015, pp. 218–230 (2015)
3. Abedin, P., Hooshmand, S., Ganguly, A., Thankachan, S.V.: The heaviest induced ancestors problem revisited. In: Annual Symposium on Combinatorial Pattern Matching, CPM 2018, July 2–4, 2018—Qingdao, China, pp. 20:1–20:13 (2018)
4. Afshani, P., Nielsen, J.S.: Data structure lower bounds for document indexing problems. In 43rd International Colloquium on Automata, Languages, and Programming, ICALP 2016, July 11–15, 2016, Rome, Italy, pp. 93:1–93:15 (2016)
5. Agarwal, P.K.: Range searching. In: Goodman, J.E., O'Rourke, J. (eds.) Handbook of Discrete and Computational Geometry, 2nd edn, pp. 809–837. Chapman and Hall, Boca Raton (2004)
6. Alstrup, S., Brodal, G.S., Rauhe, T.: New data structures for orthogonal range searching. In: *41st Annual Symposium on Foundations of Computer Science, FOCS 2000, 12-14 November 2000, Redondo Beach, California, USA*, pages 198–207, (2000)
7. Alstrup, S., Brodal, G.S., Rauhe, T.: Pattern matching in dynamic texts. In: Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA '00, pp. 819–828, Philadelphia, PA, USA, Society for Industrial and Applied Mathematics (2000)
8. Amir, A., Boneh, I.: Locally maximal common factors as a tool for efficient dynamic string algorithms. In: Annual Symposium on Combinatorial Pattern Matching, CPM 2018, July 2–4, 2018 - Qingdao, China, pp. 11:1–11:13 (2018)
9. Amir, A., Boneh, I.: Dynamic palindrome detection. CoRR (2019). arXiv:1906.09732
10. Amir, A., Boneh, I., Charalampopoulos, P., Kondratovsky, E.: Repetition detection in a dynamic string. In: 27th Annual European Symposium on Algorithms, ESA 2019, September 9–11, 2019, Munich/Garching, Germany, pp. 5:1–5:18 (2019)
11. Amir, A., Charalampopoulos, P. Iliopoulos, C.S., Pissis, S.P., Radoszewski, J.: Longest common factor after one edit operation. In: String Processing and Information Retrieval—24th International Symposium, SPIRE 2017, Palermo, Italy, September 26–29, 2017, Proceedings, pp. 14–26 (2017)
12. Amir, A., Charalampopoulos, P., Pissis, S.P., Radoszewski, J.: Longest common substring made fully dynamic. In: 27th Annual European Symposium on Algorithms, ESA 2019, September 9–11, 2019, Munich/Garching, Germany, pp. 6:1–6:17 (2019)

13. Amir, A., Landau, G.M., Lewenstein, M., Sokol, D.: Dynamic text and static pattern matching. ACM Trans. Algorithms **3**(2), 19 (2007)
14. Amir, A., Lewenstein, M., Thankachan, S.V.: Range LCP queries revisited. In: String Processing and Information Retrieval—22nd International Symposium, SPIRE 2015, London, UK, September 1–4, 2015, Proceedings, pp. 350–361 (2015)
15. Apostolico, A., Crochemore, M.: Fast parallel Lyndon factorization with applications. Math. Syst. Theory **28**(2), 89–108 (1995)
16. Apostolico, A., Crochemore, M., Farach-Colton, M., Galil, Z., Muthukrishnan, S.: Forty years of text indexing. In: Combinatorial Pattern Matching, 24th Annual Symposium, CPM 2013, Bad Herrenalb, Germany, June 17–19, 2013. Proceedings, pp. 1–10 (2013)
17. Ayad, L.A.K., Barton, C., Charalampopoulos, P., Iliopoulos, C.S., Pissis, S.P.: Longest common prefixes with k-errors and applications. In: String Processing and Information Retrieval - 25th International Symposium, SPIRE 2018, Lima, Peru, October 9–11, 2018, Proceedings, pp. 27–41 (2018)
18. Backurs, A., Indyk, P.: Edit distance cannot be computed in strongly subquadratic time (unless SETH is false). SIAM J. Comput. **47**(3), 1087–1097 (2018)
19. Bannai, H., I, T., Inenaga, S., Nakashima, Y., Takeda, M., Tsuruta, K.: A new characterization of maximal repetitions by Lyndon trees. In: Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015, San Diego, CA, USA, January 4–6, 2015, pp. 562–571 (2015)
20. Bannai, H., I, T., Inenaga, S., Nakashima, Y., Takeda, M., Tsuruta, K.: The "runs" theorem. SIAM J. Comput. **46**(5), 1501–1514 (2017)
21. Barcelo, H.: On the action of the symmetric group on the free Lie algebra and the partition lattice. J. Comb. Theory Ser. A **55**(1), 93–129 (1990)
22. Bender, M.A., Farach-Colton, M.: The LCA problem revisited. In: LATIN 2000: Theoretical Informatics, 4th Latin American Symposium, Punta del Este, Uruguay, April 10–14, 2000, Proceedings, pp. 88–94 (2000)
23. Bentley, J.L.: Multidimensional divide-and-conquer. Commun. ACM **23**(4), 214–229 (1980)
24. Borozdin, K., Kosolobov, D., Rubinchik, M., Shur, A.M.: Palindromic length in linear time. In: 28th Annual Symposium on Combinatorial Pattern Matching, CPM 2017, July 4–6, 2017, Warsaw, Poland, pp. 23:1–23:12 (2017)
25. Bringmann, K., Künnemann, M.: Quadratic conditional lower bounds for string problems and dynamic time warping. In: 56th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2015, pp. 79–97. IEEE Computer Society (2015)
26. Burkhardt, S., Kärkkäinen, J.: Fast lightweight suffix array construction and checking. In: Combinatorial Pattern Matching, 14th Annual Symposium, CPM 2003, Morelia, Michocán, Mexico, June 25–27, 2003, Proceedings, pp. 55–69 (2003)
27. Chan, T. M., Larsen, K. G., Pătraşcu, M.: Orthogonal range searching on the RAM, revisited. In: Proceedings of the 27th ACM Symposium on Computational Geometry, Paris, France, June 13–15, 2011, pp. 1–10 (2011)
28. Charalampopoulos, P., Crochemore, M., Iliopoulos, C.S., Kociumaka, T., Pissis, S.P., Radoszewski, J., Rytter, W., Waleń, T.: Linear-time algorithm for long LCF with k mismatches. In: Annual Symposium on Combinatorial Pattern Matching, CPM 2018, July 2–4, 2018 - Qingdao, China, pp. 23:1–23:16 (2018)
29. Charalampopoulos, P., Gawrychowski, P., Pokorski, K.: Dynamic longest common substring in polylogarithmic time. In: 47th International Colloquium on Automata, Languages, and Programming, ICALP 2020, pp. 27:1–27:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2020)
30. Charalampopoulos, P., Kociumaka, T., Mohamed, M., Radoszewski, J., Rytter, W., Straszyński, J., Waleń, T., Zuba, W.: Counting distinct patterns in internal dictionary matching. In: 31st Annual Symposium on Combinatorial Pattern Matching, CPM 2020, pp. 8:1–8:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2020)
31. Charalampopoulos, P., Kociumaka, T., Mohamed, M., Radoszewski, J. Rytter, W., Waleń, T.: Internal dictionary matching. In: 30th International Symposium on Algorithms and Computation, ISAAC 2019, December 8–11, 2019, Shanghai, China, pp. 22:1–22:17 (2019)
32. Charalampopoulos, P., Kociumaka, T., Mozes, S.: Dynamic string alignment. In: 31st Annual Symposium on Combinatorial Pattern Matching, CPM 2020, volume 161 of LIPIcs, pp. 9:1–9:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2020)
33. Charalampopoulos, P., Kociumaka, T., Wellnitz, P.: Faster approximate pattern matching: a unified approach. CoRR (2020). arXiv:2004.08350

34. Chen, K.-T., Fox, R.H., Lyndon, R.C.: Free differential calculus, IV. Ann. Math. **68**, 81–95 (1958)
35. Clifford, R., Grønlund, A., Larsen, K.G., Starikovskaya, T.A.: Upper and lower bounds for dynamic data structures on strings. In: 35th Symposium on Theoretical Aspects of Computer Science, STACS 2018, February 28–March 3, 2018, Caen, France, pp. 22:1–22:14 (2018)
36. Cohen, H., Porat, E.: On the hardness of distance oracle for sparse graph. CoRR (2010). arXiv:1006.1117
37. Crochemore, M., Hancart, C., Lecroq, T.: Algorithms on Strings. Cambridge University Press, Cambridge (2007)
38. Crochemore, M., Iliopoulos, C.S., Mohamed, M., Sagot, M.: Longest repeats with a block of k don't cares. Theoret. Comput. Sci. **362**(1–3), 248–254 (2006)
39. Daykin, J.W., Iliopoulos, C.S., Smyth, W.F.: Parallel RAM algorithms for factorizing words. Theoret. Comput. Sci. **127**(1), 53–67 (1994)
40. Dietz, P.F., Mehlhorn, K., Raman, R., Uhrig, C.: Lower bounds for set intersection queries. Algorithmica **14**(2), 154–168 (1995)
41. Farach, M.: 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, pp. 137–143 (1997)
42. Ferragina, P.: Dynamic text indexing under string updates. J. Algorithms **22**(2), 296–328 (1997)
43. Ferragina, P., Grossi, R.: Optimal on-line search and sublinear time update in string matching. SIAM J. Comput. **27**(3), 713–736 (1998)
44. Fici, G., Gagie, T., Kärkkäinen, J., Kempa, D.: A subquadratic algorithm for minimum palindromic factorization. J. Discrete Algorithms **28**, 41–48 (2014)
45. Fischer, J., Köppl, D., Kurpicz, F.: On the benefit of merging suffix array intervals for parallel pattern matching. In: 27th Annual Symposium on Combinatorial Pattern Matching, CPM 2016, June 27–29, 2016, Tel Aviv, Israel, pp. 26:1–26:11 (2016)
46. Flouri, T., Giaquinta, E., Kobert, K., Ukkonen, E.: Longest common substrings with k mismatches. Inf. Process. Lett. **115**(6–8), 643–647 (2015)
47. Fredman, M.L., Komlós, J., Szemerédi, E.: Storing a sparse table with O(1) worst case access time. J. ACM **31**(3), 538–544 (1984)
48. Funakoshi, M., Nakashima, Y., Inenaga, S., Bannai, H., Takeda, M.: Longest substring palindrome after edit. In: Annual Symposium on Combinatorial Pattern Matching, CPM 2018, July 2–4, 2018—Qingdao, China, pp. 12:1–12:14 (2018)
49. Funakoshi, M., Nakashima, Y., Inenaga, S., Bannai, H., Takeda, M.: Faster queries for longest substring palindrome after block edit. In: 30th Annual Symposium on Combinatorial Pattern Matching, CPM 2019, June 18–20, 2019, Pisa, Italy, pp. 27:1–27:13 (2019)
50. Gagie, T., Gawrychowski, P., Nekrich, Y.: 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 (2013)
51. Gawrychowski, P., Karczmarz, A., Kociumaka, T., Lacki, J., Sankowski, P.: Optimal dynamic strings. In: Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, New Orleans, LA, USA, January 7–10, 2018, pp. 1509–1528 (2018). Full version available at arXiv:1511.02612
52. Goldstein, I., Kopelowitz, T., Lewenstein, M., Porat, E.: Conditional lower bounds for space/time tradeoffs. In: Algorithms and Data Structures—15th International Symposium, WADS 2017, St. John's, NL, Canada, July 31–August 2, 2017, Proceedings, pp. 421–436 (2017)
53. Gu, M., Farach, M., Beigel, R.: An efficient algorithm for dynamic text indexing. In: Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA '94, pp. 697–704, Philadelphia, PA, USA, (1994)
54. Gusfield, D.: Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology. Cambridge University Press, New York (1997)
55. Hyyrö, H., Narisawa, K., Inenaga, S.: Dynamic edit distance table under a general weighted cost function. J. Discrete Algorithms **34**, 2–17 (2015)
56. I, T., Nakashima, Y., Inenaga, S., Bannai, H., Takeda, M.: Faster Lyndon factorization algorithms for SLP and LZ78 compressed text. Theoret. Comput. Sci. **656**, 215–224 (2016)
57. I, T., Sugimoto, S., Inenaga, S., Bannai, H., Takeda, M.: Computing palindromic factorizations and palindromic covers on-line. In: Combinatorial Pattern Matching—25th Annual Symposium, CPM 2014, Moscow, Russia, June 16–18, 2014. Proceedings, pp. 150–161 (2014)

58. Karp, R.M., Rabin, M.O.: Efficient randomized pattern-matching algorithms. IBM J. Res. Dev. **31**(2), 249–260 (1987)
59. Knuth, D.E.: The Art of Computer Programming, Volume 4, Fascicle 2: Generating All Tuples and Permutations. Addison-Wesley Professional, New York (2005)
60. Kociumaka, T.: Minimal suffix and rotation of a substring in optimal time. In: 27th Annual Symposium on Combinatorial Pattern Matching, CPM 2016, June 27–29, 2016, Tel Aviv, Israel, pp. 28:1–28:12, (2016)
61. Kociumaka, T.: Efficient data structures for internal queries in texts. Ph.D. thesis, University of Warsaw, Oct. 2018. https://www.mimuw.edu.pl/~kociumaka/files/phd.pdf
62. Kociumaka, T., Radoszewski, J., Rytter, W., Waleń, T.: Internal pattern matching queries in a text and applications. In: Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015, San Diego, CA, USA, January 4–6, 2015, pp. 532–551 (2015)
63. Kociumaka, T., Starikovskaya, T.A., Vildhøj, H.W.: Sublinear space algorithms for the longest common substring problem. In: Algorithms - ESA 2014 - 22th Annual European Symposium, Wroclaw, Poland, September 8–10, 2014. Proceedings, pp. 605–617 (2014)
64. Lyndon, R.C.: On Burnside's problem. Trans. Am. Math. Soc. **77**, 202–215 (1954)
65. Maekawa, M.: A $\sqrt{n}$ algorithm for mutual exclusion in decentralized systems. ACM Trans. Comput. Syst. **3**(2), 145–159 (1985)
66. Manacher, G.K.: A new linear-time "on-line" algorithm for finding the smallest initial palindrome of a string. J. ACM **22**(3), 346–351 (1975)
67. Mehlhorn, K., Sundar, R., Uhrig, C.: Maintaining dynamic sequences under equality tests in polylogarithmic time. Algorithmica **17**(2), 183–198 (1997)
68. Mucha, M.: Lyndon words and short superstrings. In: Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2013, New Orleans, Louisiana, USA, January 6–8, 2013, pp. 958–972, (2013)
69. Pătraşcu, M., Roditty, L.: Distance oracles beyond the Thorup–Zwick bound. SIAM J. Comput. **43**(1), 300–311 (2014)
70. Sahinalp, S.C., Vishkin, U.: 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, pp. 320–328 (1996)
71. Sleator, D.D., Tarjan, R.E.: A data structure for dynamic trees. J. Comput. Syst. Sci. **26**(3), 362–391 (1983)
72. Starikovskaya, T. A.: Longest common substring with approximately *k* mismatches. In: 27th Annual Symposium on Combinatorial Pattern Matching, CPM 2016, June 27–29, 2016, Tel Aviv, Israel, pp. 21:1–21:11 (2016)
73. Starikovskaya, T.A., Vildhøj, H. W.: Time-space trade-offs for the longest common substring problem. In: Combinatorial Pattern Matching, 24th Annual Symposium, CPM 2013, Bad Herrenalb, Germany, June 17–19, 2013. Proceedings, pp. 223–234 (2013)
74. Sundar, R., Tarjan, R.E.: Unique binary-search-tree representations and equality testing of sets and sequences. SIAM J. Comput. **23**(1), 24–44 (1994)
75. Thankachan, S.V., Aluru, C., Chockalingam, S.P., Aluru, S.: Algorithmic framework for approximate matching under bounded edits with applications to sequence analysis. In: Research in Computational Molecular Biology—22nd Annual International Conference, RECOMB 2018, Paris, France, April 21–24, 2018, Proceedings, pp. 211–224 (2018)
76. Thankachan, S.V., Apostolico, A., Aluru, S.: A provably efficient algorithm for the k-mismatch average common substring problem. J. Comput. Biol. **23**(6), 472–482 (2016)
77. Urabe, Y., Nakashima, Y., Inenaga, S., Bannai, H., Takeda, M.: Longest Lyndon substring after edit. In: Annual Symposium on Combinatorial Pattern Matching, CPM 2018, July 2–4, 2018—Qingdao, China, pp. 19:1–19:10 (2018)
78. Weiner, P.: Linear pattern matching algorithms. In: 14th Annual Symposium on Switching and Automata Theory, Iowa City, Iowa, USA, October 15–17, 1973, pp. 1–11 (1973)

Affiliations

Amihood Amir[1] · Panagiotis Charalampopoulos[2] · Solon P. Pissis[3,4] · Jakub Radoszewski[5,6]

Amihood Amir
amir@esc.biu.ac.il

Panagiotis Charalampopoulos
panagiotis.charalampopoulos@kcl.ac.uk

Jakub Radoszewski
jrad@mimuw.edu.pl

1 Department of Computer Science, Bar-Ilan University, Ramat Gan, Israel

2 Department of Informatics, King's College London, London, UK

3 CWI, Amsterdam, The Netherlands

4 Vrije Universiteit, Amsterdam, The Netherlands

5 Institute of Informatics, University of Warsaw, Warsaw, Poland

6 Samsung R&D Institute Poland, Warsaw, Poland