Dynamic Longest Common Substring in Polylogarithmic Time

Panagiotis Charalampopoulos
Department of Informatics, King's College London, UK
Institute of Informatics, University of Warsaw, Poland
panagiotis.charalampopoulos@kcl.ac.uk

Paweł Gawrychowski
Institute of Computer Science, University of Wrocław, Poland
gawry@cs.uni.wroc.pl

Karol Pokorski
Institute of Computer Science, University of Wrocław, Poland
pokorski@cs.uni.wroc.pl

Abstract

The longest common substring problem consists in finding a longest string that appears as a (contiguous) substring of two input strings. We consider the dynamic variant of this problem, in which we are to maintain two dynamic strings S and T, each of length at most n, that undergo substitutions of letters, in order to be able to return a longest common substring after each substitution. Recently, Amir et al. [ESA 2019] presented a solution for this problem that needs only $\tilde{\mathcal{O}}(n^{2/3})$ time per update. This brought the challenge of determining whether there exists a faster solution with polylogarithmic update time, or (as is the case for other dynamic problems), we should expect a polynomial (conditional) lower bound. We answer this question by designing a significantly faster algorithm that processes each substitution in amortized $\log^{\mathcal{O}(1)} n$ time with high probability. Our solution relies on exploiting the local consistency of the parsing of a collection of dynamic strings due to Gawrychowski et al. [SODA 2018], and on maintaining two dynamic trees with labeled bicolored leaves, so that after each update we can report a pair of nodes, one from each tree, of maximum combined weight, which have at least one common leaf-descendant of each color. We complement this with a lower bound of $\Omega(\log n/\log\log n)$ for the update time of any polynomial-size data structure that maintains the LCS of two dynamic strings, even allowing amortization and randomization.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases string algorithms, dynamic algorithms, longest common substring

Digital Object Identifier 10.4230/LIPIcs.ICALP.2020.27

Category Track A: Algorithms, Complexity and Games

Related Version A full version of the paper is available at `https://arxiv.org/abs/2006.02408`.

Funding *Panagiotis Charalampopoulos*: Partially supported by ERC grant TOTAL under the European Union's Horizon 2020 Research and Innovation Programme (agreement no. 677651).

1 Introduction

The well-known longest common substring (LCS) problem, formally stated below, was conjectured by Knuth to require $\Omega(n\log n)$ time. However, in his seminal paper that introduced suffix trees, Weiner showed how to solve it in linear time (for constant alphabets) [29]. Since then, this classical question was considered in many different versions, such as obtaining a tradeoff between the time and the working space [21, 27], or computing an approximate LCS under either the Hamming or the edit distance (see [9, 20, 28] and references therein), to name a few.

Problem: Longest Common Substring
Input: Two strings S and T of length at most n over an alphabet Σ.
Output: A longest substring X of S that is a substring of T.

We consider the dynamic version of this problem where the strings are updated and we are to report an LCS after each update. That is, we return the length of an LCS and a pair of starting positions of its occurrences in the strings. The allowed update operations are substitutions of single letters in either S or T. In fact, with due care, our algorithms can be adapted to handle all edit operations, i.e. insertions and deletions as well, but we only allow substitutions for the sake of a clearer exposition of the main ideas.

Dynamic problems on strings are of wide interest. Maybe the most basic question in this direction is that of maintaining a dynamic text while enabling efficient pattern matching queries. This is clearly motivated by, say, the possible application in a text editor. The first structure achieving polylogarithmic update time and optimal query time for this problem was designed by Sahinalp and Vishkin [26]. Later, the update time was improved to $\mathcal{O}(\log^2 n \log\log n \log^* n)$ at the cost of $\mathcal{O}(\log n \log\log n)$ additional time per query by Alstrup et al. [2]. Recently, Gawrychowski et al. [15] presented a data structure that requires $\mathcal{O}(\log^2 n)$ time per update and allows for time-optimal queries. Other problems on strings that have been studied in the dynamic setting include maintaining repetitions, such as the set of square substrings [5] or a longest palindromic substring [4, 7].

As for the LCS problem itself, Amir et al. [6] initiated the study of this question in the dynamic setting by considering the problem of constructing a data structure over two strings that returns the LCS after a single edit operation in one of the strings. However, in their solution, after each edit operation, the string is immediately reverted to its original version. Abedin et al. [1] improved the tradeoffs for this problem by designing a more efficient solution for the so-called heaviest induced ancestors problem. Amir and Boneh [3] investigated some special cases of the *partially dynamic LCS* problem (in which one of the strings is assumed to be static); namely, the case where the static string is periodic and the case where the substitutions in the dynamic string are substitutions with some letter $\# \notin \Sigma$. Finally, Amir et al. [7] presented the first algorithm for the *fully dynamic LCS* problem (in which both strings are subject to updates) that needs only sublinear time per edit operation (insertion or deletion of a letter) in either string, namely $\tilde{\mathcal{O}}(n^{2/3})$. As a stepping stone towards this result, they designed an algorithm for the partially dynamic LCS problem that takes $\tilde{\mathcal{O}}(\sqrt{n})$ time per edit operation.

For some natural dynamic problems, the best known bounds on the query and the update time are of the form $\mathcal{O}(n^\alpha)$, where n is the size of the input and α is some constant. Henzinger et al. [16] introduced the online Boolean matrix-vector multiplication conjecture that can be used to provide some justification for the polynomial time hardness of many such dynamic problems in a unified manner. This brings the question of determining if the bound on the update time in the dynamic LCS problem should be polynomial or subpolynomial.

We answer this question by significantly improving on the bounds presented by Amir et al. [7] and presenting a solution for the fully dynamic LCS problem that handles each update in amortized polylogarithmic time with high probability. As a warm-up, we present a (relatively simple) deterministic solution for the partially dynamic LCS problem that handles each update in amortized $\mathcal{O}(\log^2 n)$ time.

After having determined that the complexity of fully dynamic LCS is polylogarithmic, the next natural question is whether we can further improve the bound to polyloglogarithmic. By now we have techniques that can be used to not only distinguish between these two situations

but (in some cases) also provide tight bounds. As a prime example, static predecessor for a set of n numbers from $[n^2]$ requires $\Omega(\log\log n)$ time for structures of size $\tilde{\mathcal{O}}(n)$ [25], and dynamic connectivity for forests requires $\Omega(\log n)$ time [24], with both bounds being asymptotically tight. In some cases, seemingly similar problems might have different complexities, as in the orthogonal range emptiness problem: Nekrich [22] showed a data structure of size $\mathcal{O}(n\log^4 n)$ with $\mathcal{O}(\log^2\log n)$ query time for 3 dimensions, while for the same problem in 4 dimensions Pătraşcu showed that any polynomial-size data structure requires $\Omega(\log n/\log\log n)$ query time [23]. In the full version of this work, we show the following results, each obtained through a series of reductions, starting from the problem of answering reachability queries in butterfly graphs that was considered in the seminal paper of Pătraşcu [23].

▶ **Theorem 1.** *Any structure of $\tilde{\mathcal{O}}(n)$ size for maintaining an LCS of a dynamic string S and a static string T, each of length at most n, requires $\Omega(\log n/\log\log n)$ time per update operation.*

▶ **Theorem 2.** *Any polynomial-size structure for maintaining the LCS of two dynamic strings of length n requires $\Omega(\log n/\log\log n)$ time per update operation.*

Finally, we demonstrate that the difference in the allowed space in the above two lower bounds is indeed needed. To this end, we show that partially dynamic LCS admits an $\mathcal{O}(n^{1+\epsilon})$-space, $\mathcal{O}(\log\log n)$-update time solution, for any $\epsilon > 0$.

Techniques and roadmap. We first consider the partially dynamic version of the problem where updates are only allowed in one of the strings, say S, in Section 3. This problem is easier as we can use the static string T as a reference point. We maintain a partition of S into blocks (i.e. substrings of S whose concatenation equals S), such that each block is a substring of T, but the concatenation of any two consecutive blocks is not. This is similar to the approach of [8] and other works that consider one dynamic and one static string. The improvement upon the $\tilde{\mathcal{O}}(\sqrt{n})$-time algorithm presented in [7] comes exactly from imposing the aforementioned maximality property, which guarantees that the sought LCS is a substring of the concatenation of at most three consecutive blocks and contains the first letter of one of these blocks. The latter property allows us to anchor the LCS in S. Upon an update, we can maintain the block decomposition, by updating a constant number of blocks. It then suffices to show how to efficiently compute the longest substring of T that contains the first letter of a given block. We reduce this problem to answering a *heaviest induced ancestors* (HIA) query. This reduction was also presented in [1,6], but we describe the details to make following the more involved solution of fully dynamic LCS easier.

In Section 4 we move to the fully dynamic LCS problem. We try to anchor the LCS in both strings as follows. For each of the strings S and T we show how to maintain, in $\log^{\mathcal{O}(1)} n$ time, a collection of pairs of adjacent fragments (e.g. $(S[i\,..\,j-1], S[j\,..\,k])$), denoted by J_S for S and J_T for T, with the following property. For any common substring X of S and T there exists a partition $X = X_\ell X_r$ for which there exists a pair $(U_\ell, U_r) \in J_S$ and a pair $(V_\ell, V_r) \in J_T$ such that X_ℓ is a suffix of both U_ℓ and V_ℓ, while X_r is a prefix of both U_r and V_r. We can maintain this collection by exploiting the properties of the locally consistent parsing previously used for maintaining a dynamic collection of strings [15]. We maintain tries for fragments in the collections and reduce the dynamic LCS problem to a problem on dynamic bicolored trees, which we solve by using dynamic heavy-light decompositions and 2D range trees.

2 Preliminaries

We use $[n]$ to denote the set $\{1, 2, \ldots, n\}$. Let $S = S[1]S[2]\cdots S[n]$ be a *string* of length $|S| = n$ over an integer alphabet Σ. For two positions i and j on S, we denote by $S[i\,..\,j] = S[i]\cdots S[j]$ the *fragment* of S that starts at position i and ends at position j (it is the empty string ε if $j < i$). A string Y, of length m with $0 < m \leq n$, is a *substring* of S if there exists a position i in S such that $Y = S[i\,..\,i+m-1]$. The prefix of S ending at the i-th letter of S is denoted by $S[..\,i]$ and the suffix of S starting at the i-th letter of S is denoted by $S[i\,..]$. The reverse string of S is denoted by S^R. The concatenation of strings S and T is denoted by ST, and the concatenation of k copies of string S is denoted by S^k. By $\mathsf{lcp}(S, T)$ we denote the length of the longest common prefix of strings S and T.

We define the trie of a collection of strings $C = \{S_1, S_2, \ldots, S_k\}$ as follows. It is a rooted tree with edges labeled by single letters. Every string S that is a prefix of some string in C is represented by exactly one path from the root to some node v of the tree, such that the concatenation of the labels of the edges of the path, the *path-label* of v, is equal to S. The compacted trie of C is obtained by contracting maximal paths consisting of nodes with one child to an edge labeled by the concatenation of the labels of the edges of the path. Usually, the label of the new edge is stored as the start/end indices of the corresponding fragment of some S_i. The *suffix tree* of a string T is the compacted trie of all suffixes of $T\$$ where $\$$ is a letter smaller than all letters of the alphabet Σ. It can be constructed in $\mathcal{O}(|T|)$ time for linear-time sortable alphabets [11]. For a node u in a (compacted) trie, we define its *depth* as the number of edges on the path from the root to u. Analogously, we define the *string-depth* of u as the total length of labels along the path from the root to u.

We say that a tree is weighted if there is a weight $w(u)$ associated with each node u of the tree, such that weights along the root-to-leaf paths are increasing, i.e. for any node u other than the root, $w(u) > w(\text{parent}(u))$. Further, we say that a tree is labeled if each of its leaves is given a distinct label.

▶ **Definition 3.** *For rooted, weighted, labeled trees $\mathcal{T}_1$ and $\mathcal{T}_2$, two nodes $u \in \mathcal{T}_1$ and $v \in \mathcal{T}_2$, are* induced *(by ℓ) if and only if there are leaves x and y with the same label ℓ, such that x is a descendant of u and y is a descendant of v.*

Problem: Heaviest Induced Ancestors
Input: Two rooted, weighted, labeled trees $\mathcal{T}_1$ and $\mathcal{T}_2$ of total size n.
Query: Given a pair of nodes $u \in \mathcal{T}_1$ and $v \in \mathcal{T}_2$, return a pair of nodes u', v' such that u' is ancestor of u, v' is ancestor of v, u' and v' are induced and they have the largest total combined weight $w(u') + w(v')$.

This problem was introduced in [14], with the last advances made in [1]. The next lemma encapsulates one of the known trade-offs.

▶ **Lemma 4** ([14]). *There is a data structure for the Heaviest Induced Ancestors problem, that can be built in $\mathcal{O}(n \log^2 n)$ time and answers queries in $\mathcal{O}(\log^2 n)$ time.*

3 Partially Dynamic LCS

In this section, we describe an algorithm for solving the partially dynamic variant of the LCS problem, where updates are only allowed on one of the strings, say S, while T is given in advance and is not subject to change.

Let us assume for now that all the letters of S throughout the execution of the algorithm occur at least once in T; we will waive this assumption later. Also, for simplicity, we assume that S is initially equal to $\$^{|S|}$, for $\$ \notin \Sigma$. We can always obtain any other initial S by performing an appropriate sequence of updates in the beginning.

▶ **Definition 5.** *A* block decomposition *of string S with respect to string T is a sequence of strings $(s_1, s_2, \ldots, s_k)$ such that $S = s_1 s_2 \ldots s_k$ and every s_i is a fragment of T. An element of the sequence is called a* block *of the decomposition. A decomposition is* maximal *if and only if $s_i s_{i+1}$ is not a substring of T for every $i \in [k-1]$.*

Maximal block decompositions are not necessarily unique and may have different lengths, but all admit the following useful property.

▶ **Lemma 6.** *For any maximal block decomposition of S with respect to T, any fragment of S that occurs in T is contained in at most three consecutive blocks. Furthermore, any occurrence of an LCS of S and T in S must contain the first letter of some block.*

Proof. We prove the first claim by contradiction. If $(s_1, s_2, \ldots, s_k)$ is a maximal block decomposition of S with respect to T and a fragment of S that occurs in T spans at least four consecutive blocks $s_i, s_{i+1}, s_{i+2}, \ldots, s_j$, then $s_{i+1}s_{i+2}$ is a substring of T, a contradiction.

As for the second claim, it is enough to observe, that if an occurrence of an LCS in S starts in some other than the first position of a block b, then it must contain the first letter of the next block, as otherwise its length would be smaller than the length of block b, which is a common substring of S and T. ◀

We will show that an update in S can be processed by considering a constant number of blocks in a maximal block decomposition of S with respect to T. We first summarize the basic building block needed for efficiently maintaining such a maximal block decomposition.

▶ **Lemma 7.** *Let T be a string of length at most n. After $\mathcal{O}(n \log^2 n)$-time and $\mathcal{O}(n)$-space preprocessing, given two fragments U and V of T, one can compute a longest fragment of T that is equal to a prefix of UV in $\mathcal{O}(\log \log n)$ time.*

Proof. We build a weighted ancestor queries structure over the suffix tree of T. A weighted ancestor query (ℓ, u) on a (weighted) tree $\mathcal{T}$, asks for the deepest ancestor of u with weight at most ℓ. Such queries can be answered in $\mathcal{O}(\log \log n)$ time after an $\mathcal{O}(n)$-time preprocessing of $\mathcal{T}$ if all weights are polynomial in n [12], as is the case for suffix trees with the weight of each node being its string-depth. We also build a data structure for answering unrooted LCP queries over the suffix tree of T. In our setting, such queries can be defined as follows: given nodes u and v of the suffix tree of T, we want to compute the (implicit or explicit) node where the search for the path-label of v starting from node u ends. Cole et al. [10] showed how to construct in $\mathcal{O}(n \log^2 n)$ time a data structure of size $\mathcal{O}(n \log n)$ that answers unrooted LCP queries in $\mathcal{O}(\log \log n)$ time. With these data structures at hand, the longest prefix of UV that is a fragment of T can be computed as follows. First, we retrieve the nodes of the suffix tree of T corresponding to U and V using weighted ancestor queries in $\mathcal{O}(\log \log n)$ time. In more detail, if $U = T[i\,..\,j]$ then we access the leaf of the suffix tree corresponding to $T[i\,..]$ and access its ancestor at string-depth $|U|$, and similarly for V. Second, we ask an unrooted LCP query to obtain the node corresponding to the sought prefix of UV. ◀

▶ **Lemma 8.** *A maximal block decomposition of a dynamic string S, with respect to a static string T, can be maintained in $\mathcal{O}(\log \log n)$ time per substitution operation with a data structure of size $\mathcal{O}(n \log n)$ that can be constructed in $\mathcal{O}(n \log^2 n)$ time.*

Proof. We keep the blocks on a doubly-linked list and we store the starting positions of blocks in an $\mathcal{O}(n)$-size predecessor/successor data structure over $[n]$ that supports $\mathcal{O}(\log\log n)$-time queries and updates [30]. This allows us to navigate in the structure of blocks, and in particular to be able to compute the block in which the edit occurred and its neighbors.

Suppose that we have a maximal block decomposition $B = (s_1, \ldots, s_k)$ of S with respect to T. Consider an operation which updates the letter x located in block s_i to y, so that $s_i = s_i^{\mathrm{l}} x s_i^{\mathrm{r}}$. Consider a block decomposition $B' = (s_1, s_2, \ldots, s_{i-1}, s_i^{\mathrm{l}}, y, s_i^{\mathrm{r}}, s_{i+1}, \ldots, s_k)$ of string S' after the update. Note that both s_i^{l} and s_i^{r} may be empty. This block decomposition does not need to be maximal. However, since B is a maximal block decomposition of S, none of the strings s_1s_2, s_2s_3, …, $s_{i-2}s_{i-1}$, $s_{i+1}s_{i+2}$, $s_{i+2}s_{i+3}$, …, $s_{k-1}s_k$ occurs in T. Thus, given B', we repeatedly merge any two consecutive blocks from $(s_{i-1}, s_i^{\mathrm{l}}, y, s_i^{\mathrm{r}}, s_{i+1})$ whose concatenation is a substring of T into one, until this is no longer possible. We have at most four merges before obtaining a maximal block decomposition B' of string S'. Each merge is implemented with Lemma 7 in $\mathcal{O}(\log\log n)$ time. ◀

As for allowing substitutions of letters that do not occur in T, we simply allow blocks of length 1 that are not substrings of T in block decompositions, corresponding to such letters. It is readily verified that all the statements above still hold.

Due to Lemma 6, for a maximal block decomposition $(s_1, s_2, \ldots, s_k)$ of S with respect to T, we know that any occurrence of an LCS of S and T in S must contain the first letter of some block of the decomposition and cannot span more than three blocks. In other words, it is the concatenation of a potentially empty suffix of $s_{i-1}s_i$ and a potentially empty prefix of $s_{i+1}s_{i+2}$ for some $i \in [k]$ (for convenience we consider the non-existent s_is to be equal to ε). We call an LCS that can be decomposed in such way a candidate of s_i. Our goal is to maintain the candidate proposed by each s_i in a max-heap with the length as the key. We also store a pointer to it from block s_i. The max-heap is implemented with an $\mathcal{O}(n)$-size predecessor/successor data structure over $[n]$ that supports $\mathcal{O}(\log\log n)$-time queries and updates [30]. We assume that each block s_i stores a pointer to its candidate in the max-heap.

After an update, the candidate of each block b that satisfies the following two conditions remains unchanged: (a) b did not change and (b) neither of b's neighbors at distance at most 2 changed. For the $\mathcal{O}(1)$ blocks that changed, we proceed as follows. First, in $\mathcal{O}(\log\log n)$ time, we remove from the max-heap any candidates proposed by the deleted blocks or blocks whose neighbors at distance at most 2 have changed. Then, for each new block and for each block whose neighbors at distance at most 2 have changed, we compute its candidate and insert it to the max-heap. To compute the candidate of a block s_i, we proceed as follows. We first compute the longest suffix U of $s_{i-1}s_i$ and the longest prefix V of $s_{i+1}s_{i+2}$ that occur in T in $\mathcal{O}(\log\log n)$ time using Lemma 7. Then, the problem in scope can be restated as follows: given two fragments U and V of T compute the longest fragment of UV that occurs in T. This problem can be reduced to a single HIA query over the suffix trees of T and T^R as shown in [1,6] and we provide a brief overview at the end of this section. Combining the above discussion with Lemmas 4 and 8 we obtain that an LCS can be maintained after an $\mathcal{O}(n\log^2 n)$ time preprocessing in $\mathcal{O}(\log^2 n)$ time per update. In fact, the bottleneck in the update time in this approach is in Lemma 4, that is, the HIA structure, as the additional time in the update is only $\mathcal{O}(\log\log n)$. We can thus obtain a faster data structure at the expense of slower preprocessing using the following lemma.

▶ **Lemma 9.** *For any $\varepsilon > 0$, there is a structure for the* Heaviest Induced Ancestors *problem, that can be built in $\mathcal{O}(n^{1+\varepsilon})$ time and answers queries in constant time.*

Proof. Consider an instance of HIA on two trees $\mathcal{T}_1$ and $\mathcal{T}_2$ of total size m containing at most ℓ leaves, and let b be a parameter to be chosen later. We will show how to construct a structure of size $\mathcal{O}(b^2m)$ that allows us to reduce in constant time a query concerning two nodes $u \in \mathcal{T}_1$ and $v \in \mathcal{T}_2$ to two queries to smaller instances of HIA. In each of the smaller instances the number of leaves will shrink by a factor of b, and the total size of all smaller instances will be $\mathcal{O}(m)$. Let $b = n^\delta$, where n is the total size of the original trees. We recursively repeat the construction always choosing b according to the formula. Because the depth of the recursion is at most $\log_b n = \mathcal{O}(1)$, this results in a structure of total size $\mathcal{O}(n^{1+\varepsilon})$ for $\varepsilon = 2\delta$ and allows us to answer any query in a constant number of steps, each taking constant time.

We select b evenly-spaced (in the order of in-order traversal) leaves of $\mathcal{T}_1$ and $\mathcal{T}_2$ and call them marked. Consider a query concerning a pair of nodes $u \in \mathcal{T}_1$ and $v \in \mathcal{T}_2$. Let u'', v'' be the nearest ancestors of u and v, respectively, that contain at least one marked leaf in their subtrees. u'' and v'' can be preprocessed in $\mathcal{O}(m)$ space and accessed in constant time. We have three possibilities concerning the sought ancestors u', v':

1. u' is an ancestor of u'' and v' is an ancestor of v'' (not necessarily proper),
2. u' is a descendant of u'',
3. v' is a descendant of v''.

To check the first possibility, we preprocess every pair of marked leaves x, y. Both u'' and v'' store pointers to some marked leaves in their subtrees, so it is enough to consider a query concerning two ancestors of marked leaves x, y. This can be solved similarly as preprocessing two heavy paths for HIA queries in $\mathcal{O}(\log^2 n)$ time [14], except that now we can afford to preprocess the predecessor for every possible depth on both paths in $\mathcal{O}(m)$ space, which decreases the query time to constant. The overall space is $\mathcal{O}(b^2m)$.

The second and the third possibility are symmetric, so we focus on the second. By removing all marked leaves and their ancestors from $\mathcal{T}_1$ we obtain a collection of smaller trees, each containing less than n/b leaves. Because u' is below u'', u and u' belong to the same smaller tree. For technical reasons, we want to work with $\mathcal{O}(b)$ smaller trees, so we merge all smaller trees between two consecutive marked leaves by adding the subtree induced by their roots in $\mathcal{T}_1$. Now consider the smaller tree $\mathcal{T}_1^i$ containing u (and, by assumption, also u''). We extract the subtree of $\mathcal{T}_2$ induced by the leaves of $\mathcal{T}_1^i$, call it $\mathcal{T}_2^i$, and build a smaller instance of HIA for $\mathcal{T}_1^i$ and $\mathcal{T}_2^i$. To query the smaller instance, we need to replace v by its nearest ancestor that belong to $\mathcal{T}_2^i$. This can be preprocessed for each i and v in $\mathcal{O}(bm)$ space. By construction, $\mathcal{T}_1^i$ and $\mathcal{T}_2^i$ contain less than n/b leaves, and each node of $\mathcal{T}_1$ shows up in at most two trees $\mathcal{T}_1^i$. Each node of $\mathcal{T}_2$ might appear in multiple trees $\mathcal{T}_2^i$, but the number of non-leaf nodes in $\mathcal{T}_2^i$ is smaller than its number of leaves, so the overall number of non-leaf nodes is smaller than m, and consequently the overall number of nodes is smaller than $2m$.

The construction time can be verified to be at most the size of the structure. ◀

▶ **Theorem 10.** *It is possible to maintain an LCS of a dynamic string S and a static string T, each of length at most n, (i) after an $\mathcal{O}(n \log^2 n)$-time preprocessing in $\mathcal{O}(\log^2 n)$ time per substitution operation, or (ii) after an $\mathcal{O}(n^{1+\varepsilon})$-time preprocessing in $\mathcal{O}(\log \log n)$ time per substitution operation.*

We now briefly explain the reduction to HIA in the interests of self-containment and developing intuition in a relatively easier setting before we move on to the harder problem of maintaining an LCS of two dynamic strings.

Let $\mathcal{T}_1$ and $\mathcal{T}_2$ be the suffix trees of $T\$$ and $T^R\#$, respectively, where $\$$ and $\#$ are sentinel letters not in the alphabet and lexicographically smaller than all other letters. Note that each suffix of $T\$$ corresponds to a leaf in $\mathcal{T}_1$; similarly for $\mathcal{T}_2$. We label a leaf v of $\mathcal{T}_1$ with the starting position of the suffix of $T\$$ that it represents. For $\mathcal{T}_2$, however, we label the leaf corresponding to $T^R[i\,..]\#$ with $n-i+2$. Intuitively, if we consider a split $T = T[..i-1]T[i\,..]$, the leaves corresponding to $T[i\,..]\$$ in $\mathcal{T}_1$ and $T[..i-1]^R\#$ in T_2 get the same label. Further, let the weight of each node in $\mathcal{T}_1$ and $\mathcal{T}_2$ be its string-depth. Upon query, we first compute the node p corresponding to V in $\mathcal{T}_1$ and the node q corresponding to U^R in $\mathcal{T}_2$ using weighted ancestor queries in $\mathcal{O}(\log\log n)$ time. Then the length of the longest substring of UV is exactly the sum of the weights of the nodes returned by a HIA query for p and q. (Some technicalities arise when p or q are implicit nodes, which can be overcome straightforwardly.)

4 Fully Dynamic LCS

In this section, we prove our main result.

▶ **Theorem 11.** *We can maintain an LCS of two dynamic strings, each of length at most n, in $\log^{\mathcal{O}(1)} n$ time per substitution operation.*

We start with some intuition. Let us suppose that we can maintain a decomposition of each string in blocks of length roughly 2^k for each level $k = 0, 1, \ldots, \log n$ with the following property: any two equal fragments $U = S[i\,..\,j]$ and $V = T[i'\,..\,j']$ are "aligned" by a pair of equal blocks B_1 in S and B_2 in T at some level k such that $2^k = \Theta(|U|)$. In other words, the decomposition of U (resp. V) at level k consists of a constant number of blocks, where the first and last blocks are potentially trimmed, including B_1 (resp. B_2), and the distance of the starting position of B_1 from position i in S equals the distance of the starting position of B_2 from position i' in T. The idea is that we can use such blocks as anchors for the LCS. For each level, for each string B appearing as a block in this level, we would like to design a data structure that:

a) supports insertions/deletions of strings corresponding to sequences of a constant number of level-k blocks, each containing a specified block equal to B and a boolean variable indicating the string this sequence originates from (S or T), and

b) can return the longest common substring among pairs of elements originating from different strings that is aligned by a pair of specified blocks (that are equal to B).

For each substitution in either of the strings, we would only need to update $\mathcal{O}(\log n)$ entries in our data structures – a constant number of them per level.

Unfortunately, it is not clear how to maintain a decomposition with these properties. We resort to the dynamic maintenance of a *locally consistent parsing* of the two strings, due to Gawrychowski et al. [15]. We exploit the structure of this parsing in order to apply the high-level idea outlined above in a much more technically demanding setting.

4.1 Locally Consistent Parsing

The authors of [15] settled the time complexity of maintaining a collection of strings $\mathcal{W}$ under the following operations: $\mathsf{makestring}(W)$ (insert a non-empty string W), $\mathsf{concat}(W_1, W_2)$ (insert W_1W_2 to $\mathcal{W}$, for $W_1, W_2 \in \mathcal{W}$), $\mathsf{split}(W, i)$ (split the string W at position i and insert both resulting strings to $\mathcal{W}$, for $W \in \mathcal{W}$), $\mathsf{lcp}(W_1, W_2)$ (return the length of the longest common prefix of W_1 and W_2, for $W_1, W_2 \in \mathcal{W}$). Let us note that operations concat and split do not remove their arguments from $\mathcal{W}$. A substitution can be implemented with a constant number of calls to such operations.

▶ **Theorem 12** (Gawrychowski et al. [15]). *A collection $\mathcal{W}$ of strings of total length n can be dynamically maintained under operations* $\mathsf{makestring}(W)$, $\mathsf{concat}(W_1, W_2)$, $\mathsf{split}(W, i)$, *and* $\mathsf{lcp}(W_1, W_2)$ *with the operations requiring time $\mathcal{O}(\log n + |W|)$, $\mathcal{O}(\log n)$, $\mathcal{O}(\log n)$ worst-case time with high probability and $\mathcal{O}(1)$ worst-case time, respectively.*

At the heart of Theorem 12 lies a locally consistent parsing of the strings in the collection that can be maintained efficiently while the strings undergo updates. It can be interpreted as a dynamic version of the recompression method of Jeż [18, 19] (see also [17]) for a static string T. As such, we first describe the parsing of Theorem 12 for a static string T and then extend the description to the dynamic variant for a collection of strings.

A *run-length straight line program* (*RLSLP*) is a context-free grammar which generates exactly one string and contains two kinds of non-terminals: *concatenations* with production rule of the form $A \to BC$ (for symbols B, C) and *powers* with production rule of the form $A \to B^k$ (for a symbol B and an integer $k \geq 2$), where a *symbol* can be a non-terminal or a letter in Σ. Every symbol A generates a unique string denoted by $\mathsf{gen}(A)$.

Let $T = T_0$. We can compute strings $T_1, \ldots, T_H$, where $H = \mathcal{O}(\log n)$ and $|T_H| = 1$ in $\mathcal{O}(n)$ time using interleaved calls to the following two auxiliary procedures:

RunCompress applied if h is even: for each B^r, $r > 1$, replace all occurrences of B^r as a run by a new letter A. There are no runs after an application of this procedure.[1]

HalfCompress applied if h is odd: first partition Σ into Σ_ℓ and Σ_r; then, for each pair of letters $B \in \Sigma_\ell$, $C \in \Sigma_r$ such that BC occurs in T_h replace all occurrences of BC by a new letter A.

We can interpret strings $T = T_0, T_1, \ldots, T_H$ as an *uncompressed parse tree* $\mathsf{PT}(T)$, by considering their letters as nodes, so that the parent of $T_h[i]$ is the letter of T_{h+1} that either (a) corresponds to $T_h[i]$ or (b) replaced a fragment of T_h containing $T_h[i]$. We say that the node representing $T_h[i]$ is the node left (resp. right) of the node representing $T_h[i+1]$ (resp. $T_h[i-1]$). Every node v of $\mathsf{PT}(T)$ is labeled with the symbol it represents, denoted by $\mathcal{L}(v)$. For a node v corresponding to a letter of T_h, we say that the level of v, denoted by $\mathsf{lev}(v)$, is h. The *value* $\mathsf{val}(v)$ of a node v is defined as the fragment of T corresponding to the leaf descendants of v and it is an occurrence of $\mathsf{gen}(A)$ for $A = \mathcal{L}(v)$.

We define a *layer* to be any sequence of nodes $v_1 v_2 \cdots v_r$ in $\mathsf{PT}(T)$ whose values are consecutive fragments of T, i.e. $\mathsf{val}(v_j) = T[r_{j-1} + 1 \,..\, r_j]$ for some increasing sequence of r_i's. The value of a layer C is the concatenation of the values of its elements and is denoted by $\mathsf{val}(C)$. We similarly use $\mathsf{gen}(\cdot)$ for sequences of symbols, to denote the concatenation of the strings generated by them. We call a layer $v_1 v_2 \cdots v_r$ an *up-layer* when $\mathsf{lev}(v_i) \leq \mathsf{lev}(v_{i+1})$ for all i, and a *down-layer* when $\mathsf{lev}(v_i) \geq \mathsf{lev}(v_{i+1})$ for all i.

In [15], the authors show how to maintain an RLSLP for each string in the collection, each with at most $c \log n$ levels for some global constant c with high probability. Let T be a string in the collection. For each fragment $U = T[a \,..\, b]$ of T, one can compute in $\mathcal{O}(\log n)$ time a *context insensitive decomposition* that consists in a layer $\mathsf{C}(U)$ of nodes in $\mathsf{PT}(T)$ with value $T[a \,..\, b]$ and has the following property. It can be decomposed into an up-layer $\mathsf{C}_{\mathsf{up}}(U)$ and a down-layer $\mathsf{C}_{\mathsf{down}}(U)$ such that:

- The sequence of the labels of the nodes in $\mathsf{C}_{\mathsf{up}}(U)$ can be expressed as a sequence of at most $c \log n$ symbols and powers of symbols $\mathsf{d}_{\mathsf{up}}(U) = A_0^{r_0} A_1^{r_1} \cdots A_m^{r_m}$ such that, for all i, $A_i^{r_i}$ corresponds to r_i consecutive nodes at level i of $\mathsf{PT}(T)$; r_i can be 0 for $i < m$.

[1] A fragment $T[i \,..\, j] = B^r$ is a run if it is a maximal fragment consisting of Bs.

- Similarly, the sequence of the labels of the nodes in $\mathsf{C}_{\mathsf{down}}(U)$ can be expressed as a sequence of at most $c \log n$ symbols and powers of symbols $\mathsf{d}_{\mathsf{down}}(U) = B_m^{t_m} B_{m-1}^{t_{m-1}} \cdots B_0^{t_0}$ such that, for all i, $B_i^{t_i}$ corresponds to t_i consecutive nodes at level i of $\mathsf{PT}(T)$; t_i can be equal to 0.

We denote by $\mathsf{d}(U)$ the concatenation of $\mathsf{d}_{\mathsf{up}}(U)$ and $\mathsf{d}_{\mathsf{down}}(U)$. Note that $U = \mathsf{gen}(\mathsf{d}(U)) = \mathsf{gen}(A_0)^{r_0} \cdots \mathsf{gen}(A_m)^{r_m} \mathsf{gen}(B_m)^{t_m} \cdots \mathsf{gen}(B_0)^{t_0}$. See Figure 1 for a visualization. The parsing of the strings enjoys local consistency in the following way: $\mathsf{d}(U) = \mathsf{d}(V)$ for any fragment V of any string in the collection such that $U = V$. We will slightly abuse notation and use the term "context insensitive decomposition" to refer to both $\mathsf{d}(U)$ and $\mathsf{C}(U)$. In addition, we also use $\mathsf{d}(\cdot)$ for substrings and not just for fragments.

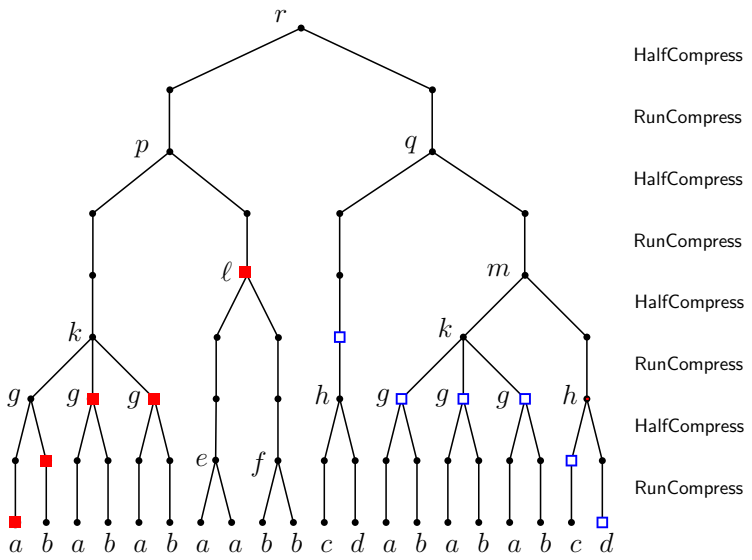

Figure 1 An example $\mathsf{PT}(T)$ for $T = T_0 = abababaabbcdababab cd$. We omit the label of each node v with a single child u; $\mathcal{L}(v) = \mathcal{L}(u)$. $T_3 = kefhkh$ and $T_6 = pq$. We denote the nodes $\mathsf{C}_{\mathsf{up}}(T)$ by red (filled) squares and the nodes of $\mathsf{C}_{\mathsf{down}}(T)$ with blue (unfilled) squares. $\mathsf{d}_{\mathsf{up}}(T) = abg^2\ell$, $\mathsf{d}_{\mathsf{down}}(T) = hg^3cd$ and hence $\mathsf{d}(T) = abg^2\ell hg^3cd$.

Let us consider any sequence of nodes corresponding, for some $j < m$, to $A_j^{r_j}$ with $r_j > 1$ or $B_j^{t_j}$ with $t_j > 1$. We note that T_j must have been obtained from T_{j-1} by an application of HalfCompress, since there are no runs after an application of procedure RunCompress. Thus, at level $j+1$ in $\mathsf{PT}(T)$, i.e. the one corresponding to T_{j+1}, all of these nodes collapse to a single one: their parent in $\mathsf{PT}(T)$. Hence, we have the following lemma.

▶ **Lemma 13.** *Let U be a fragment of T with $\mathsf{d}_{\mathsf{up}}(U) = A_0^{r_0} A_1^{r_1} \cdots A_m^{r_m}$ and $\mathsf{d}_{\mathsf{down}}(U) = B_m^{t_m} B_{m-1}^{t_{m-1}} \cdots B_0^{t_0}$. Then we have the following:*

- *The value of $\mathsf{C}_{\mathsf{up}}(U)$ is a suffix of the value of a layer L_{up} of (at most) $c \log n + r_m - 1$ level-m nodes, such that the two layers have the same rightmost node. The last r_m nodes are consecutive siblings with label A_m.*
- *The value of $\mathsf{C}_{\mathsf{down}}(U)$ is a prefix of the value of the layer L_{down} consisting of the subsequent (at most) $c \log n + \max(t_m - 1, 0)$ level-m nodes. If $t_m \neq 0$, then the first t_m nodes of L_{down} are consecutive siblings with label $B_m \neq A_m$.*

The parse trees of the strings in the collection are not maintained explicitly. However, we have access to the following pointers and functions, among others, which allow us to efficiently navigate through them. First, we can get a pointer to the root of $\mathsf{PT}(T)$ for any string T in the collection. Given a pointer P to some node v in $\mathsf{PT}(T)$ we can get $\deg(v)$ and pointers to the parent of v, the k-th child of v and the nodes to the left/right of v.

Let us now briefly explain how the dynamic data structure of [15] processes a substitution in T at some position i, that yields a string T'. First, the context insensitive decompositions of $T[..i-1]$ and $T[i+1..]$ are retrieved. These, together with the new letter at position i form a layer of $\mathsf{PT}(T')$. The sequence of the labels of the nodes of this layer can be expressed as a sequence of $\mathcal{O}(\log n)$ symbols and powers of symbols. Then, only the portion of $\mathsf{PT}(T)$ that lies above this layer needs to be (implicitly) computed, and the authors of [15] show how to do this in $\mathcal{O}(\log n)$ time. In total, we get $\mathsf{PT}(T')$ from $\mathsf{PT}(T)$ through $\mathcal{O}(\log^2 n)$ insertions and deletions of nodes and layers that consist of consecutive siblings.

4.2 Anchoring the LCS

We will rely on Lemma 13 in order to identify an LCS $S[i..j] = T[i'..j']$ at a pair of topmost nodes of the context insensitive decompositions of $S[i..j]$ and $T[i'..j']$ in $\mathsf{PT}(S)$ and $\mathsf{PT}(T)$, respectively. In order to develop some intuition, let us first sketch a solution for the case that $\mathsf{PT}(S)$ and $\mathsf{PT}(T)$ do not contain any power symbols throughout the execution of our algorithm. For each node v in one of the parse trees, let $\mathsf{Z}_\ell(v)$ be the value of the layer consisting of the (at most) $c\log n$ level-$\mathsf{lev}(v)$ nodes, with v being the layer's rightmost node, and $\mathsf{Z}_\mathsf{r}(v)$ be the value of the layer consisting of the (at most) $c\log n$ subsequent level-$\mathsf{lev}(v)$ nodes. Now, consider a common substring X of S and T and partition it into the prefix $X_\ell = \mathsf{gen}(\mathsf{d}_\mathsf{up}(X))$ and the suffix $X_r = \mathsf{gen}(\mathsf{d}_\mathsf{down}(X))$. For any fragment U of S that equals X, $\mathsf{C}_\mathsf{up}(U)$ is an up-layer of the form $v_1 \cdots v_m$. Hence, by Lemma 13, X_ℓ is a suffix of $\mathsf{Z}_\ell(v_m)$. Similarly, X_r is a prefix of $\mathsf{Z}_\mathsf{r}(v_m)$. Thus, it suffices to maintain pairs $(\mathsf{Z}_\ell(v), \mathsf{Z}_\mathsf{r}(v))$ for all nodes v in $\mathsf{PT}(S)$ and $\mathsf{PT}(T)$, and, in particular, a pair of nodes $u \in \mathsf{PT}(S)$ and $v \in \mathsf{PT}(T)$ that maximizes $\mathsf{lcp}(\mathsf{Z}_\ell(u)^R, \mathsf{Z}_\ell(v)^R) + \mathsf{lcp}(\mathsf{Z}_\mathsf{r}(u), \mathsf{Z}_\mathsf{r}(v))$. The existence of power symbols poses some technical challenges which we overcome below.

For each node of $\mathsf{PT}(T)$, we consider at most one pair consisting of an up-layer and a down-layer. The treatment of nodes differs, based on their parent. We have two cases.

1. For each node z with $\deg(z) = 2$ and $\mathcal{L}(z)$ being a concatenation symbol, for each child v of z, we consider the following layers:
   - The up-layer $\mathsf{J}_\mathsf{up}(v)$ of the (at most) $c\log n$ level-$\mathsf{lev}(v)$ consecutive nodes of $\mathsf{PT}(T)$ with v a rightmost node.
   - The down-layer $\mathsf{J}_\mathsf{down}(v)$ of the (at most) p level-$\mathsf{lev}(v)$ subsequent level-$\mathsf{lev}(v)$ nodes of $\mathsf{PT}(T)$. If the node to the right of v is a child of a node w with more than two children, then $p = c\log n + \deg(w)$. Otherwise $p = c\log n$.
2. For each node z of $\mathsf{PT}(T)$ whose label is a power symbol and has more than one child, we will consider $\mathcal{O}(\log n)$ pairs of layers. In particular, for each v, being one of the $c\log n + 1$ leftmost or $c\log n + 1$ rightmost children of z, we consider the following layers:
   - The up-layer $\mathsf{J}_\mathsf{up}(v)$ defined as the concatenation of (a) the (at most) $c\log n$ level-$\mathsf{lev}(v)$ consecutive nodes of $\mathsf{PT}(T)$ preceding the leftmost child of z and (b) all the children of z that lie weakly to the left of v, i.e. including v.
   - The down-layer $\mathsf{J}_\mathsf{down}(v)$ of the (at most) $c\log n$ subsequent level-$\mathsf{lev}(v)$ nodes of $\mathsf{PT}(T)$ – with one exception. If v is the rightmost child of z and the node to its right is a child of a node w with more than two children, then $\mathsf{J}_\mathsf{down}(v)$ consists of the $c\log n + \deg(w)$ subsequent level-$\mathsf{lev}(v)$ nodes.

In particular, we create at most one pair $(\mathsf{J}_{\mathsf{up}}(v), \mathsf{J}_{\mathsf{down}}(v))$ of layers for each node v of $\mathsf{PT}(T)$. Let $\mathsf{Y}_\ell(v) = \mathsf{val}(\mathsf{J}_{\mathsf{up}}(v))$ and $\mathsf{Y}_\mathsf{r}(v) = \mathsf{val}(\mathsf{J}_{\mathsf{down}}(v))$. Given a pointer to a node z in $\mathsf{PT}(T)$, we can compute the indices of the fragments corresponding to those layers with straightforward use of the pointers at hand in $\mathcal{O}(\log n)$ time. With a constant number of split operations, we can then add the string $\mathsf{Y}_\mathsf{r}(v)$ to our collection within $\mathcal{O}(\log n)$ time. Similarly, if we also maintain T^R in our collection of strings, we can add the reverse of $\mathsf{Y}_\ell(v)$ to the collection within $\mathcal{O}(\log n)$ time. We maintain pointers between v and these strings. Note that each node of $\mathsf{PT}(T)$ takes part in $\mathcal{O}(\log n)$ pairs of layers and these pairs can be retrieved in $\mathcal{O}(\log n)$ time. Similarly, for each node whose label is a power symbol, subsets of its children appear in $\mathcal{O}(\log n)$ pairs of layers; these can also be retrieved in $\mathcal{O}(\log n)$ time. Thus, throughout the updates on T, which delete/insert $\mathcal{O}(\log^2 n)$ nodes and layers of consecutive siblings, we can maintain the pairs of layers in $\tilde{\mathcal{O}}(1)$ time. These pairs of layers (or rather the pairs of their corresponding strings maintained in a dynamic collection) will be stored in an abstract structure presented in the next section. In order to keep the space occupied by our data structure $\tilde{\mathcal{O}}(n)$, after every n updates to the collection we delete our data structure, and initialize a new instance of it for an empty collection, on which we call $\mathsf{makestring}(S)$ and $\mathsf{makestring}(T)$. The cost of this reinitialization can be deamortized using standard techniques. We summarize the above discussion in the following lemma.

▶ **Lemma 14.** *We can maintain pairs $(\mathsf{Y}_\ell(v)^R, \mathsf{Y}_\mathsf{r}(v))$ for all v in $\mathsf{PT}(T)$ and $\mathsf{PT}(S)$, with each string given as a handle from the dynamic collection, in $\tilde{\mathcal{O}}(1)$ time per substitution, using $\tilde{\mathcal{O}}(n)$ space.*

▶ Remark 15. Note that the above lemma holds in the case that insertions and deletions are also allowed in S and T, as each such update operation is processed similarly to substitution and affects $\tilde{\mathcal{O}}(1)$ pairs $(\mathsf{Y}_\ell(v)^R, \mathsf{Y}_\mathsf{r}(v))$. Everything that follows in this section is oblivious to the kind of operations allowed in S and T.

The following lemma gives us an anchoring property, which is crucial for our approach.

▶ **Lemma 16.** *For any common substring X of S and T, there exists a partition $X = X_\ell X_r$ for which there exist nodes $u \in \mathsf{PT}(S)$ and $v \in \mathsf{PT}(T)$ such that:*
1. *X_ℓ is a suffix of $\mathsf{Y}_\ell(u)$ and $\mathsf{Y}_\ell(v)$, and*
2. *X_r is a prefix of $\mathsf{Y}_\mathsf{r}(u)$ and $\mathsf{Y}_\mathsf{r}(v)$.*

Proof. Let $\mathsf{d}_{\mathsf{up}}(X) = A_0^{r_0} A_1^{r_1} \cdots A_m^{r_m}$ and $\mathsf{d}_{\mathsf{down}}(X) = B_m^{t_m} B_{m-1}^{t_{m-1}} \cdots B_0^{t_0}$.

▷ Claim 17. Either $r_m > 1$, $t_m = 0$ and $\mathsf{gen}(\mathsf{d}_{\mathsf{up}}(X))$ is not a suffix of $A_m^{c \log n + r_m}$ or there exists a node $v \in \mathsf{PT}(T)$ such that:
1. $\mathsf{gen}(\mathsf{d}_{\mathsf{up}}(X))$ is a suffix of $\mathsf{Y}_\ell(v)$, and
2. $\mathsf{gen}(\mathsf{d}_{\mathsf{down}}(X))$ is a prefix of $\mathsf{Y}_\mathsf{r}(v)$.

Proof. We assume that $r_m = 1$ or $\mathsf{gen}(\mathsf{d}_{\mathsf{up}}(X))$ is a suffix of $A_m^{c \log n + r_m}$ or $t_m \neq 0$ and distinguish between the following cases.

Case 1. There exists an occurrence Y of X in T, where the label of the parent of the rightmost node u of $\mathsf{C}_{\mathsf{up}}(Y)$ is not a power symbol. (In this case $r_m = 1$.) Recall here, that we did not construct any pairs of layers for nodes whose parent has a single child. Let v be the highest ancestor of u with label A_m. If $u \neq v$ then all nodes that are descendants of v and strict ancestors of u have a single child, while the parent of v does not. In addition, the label of the parent of v must be a concatenation symbol, since only new letters are introduced at each level and thus we cannot have new nodes with label A_m appearing to the left/right of any strict ancestor of u. Finally, note that a layer of k level-$\mathsf{lev}(v)$ nodes with v a leftmost

(resp. rightmost) node contains an ancestor of each of the nodes in a layer of k level-$\mathsf{lev}(u)$ nodes with u a leftmost (resp. rightmost) node. Thus, an application of Lemma 13 for u straightforwardly implies our claim for v.

Case 2. There exists an occurrence Y of X in T, where the label of the parent z of the rightmost node u of $\mathsf{C}_{\mathsf{up}}(Y)$ is a power symbol. Let W be the rightmost occurrence of X in T such that the rightmost node w of $\mathsf{C}_{\mathsf{up}}(W)$ is a child of z. We have three subcases.

a) We first consider the case $r_m = 1$. Let us assume towards a contradiction that u is not one of the $c\log n+1$ leftmost or the $c\log n+1$ rightmost children of z. Then, by Lemma 13 we have that $\mathsf{gen}(\mathsf{d}_{\mathsf{up}}(X))$ is a suffix of $A_m^{c\log n}$ and $\mathsf{gen}(\mathsf{d}_{\mathsf{down}}(X))$ is a prefix of $A_m^{c\log n}$. Hence, there is another occurrence of X $|\mathsf{gen}(A_m)|$ positions to the right of Y, contradicting our assumption that Y is a rightmost occurrence.

b) In the case that $t_m \neq 0$, u must be the rightmost child of z since $A_m \neq B_m$.

c) In the remaining case that $\mathsf{gen}(\mathsf{d}_{\mathsf{up}}(X))$ is a suffix of $A_m^{c\log n+r_m}$, either $t_m > 0$ and we are done, or $\mathsf{gen}(\mathsf{C}_{\mathsf{down}}(Y))$ is a prefix of the value of the (at most) $c\log n$ level-m nodes to the right of u. In the latter case, either u is already among the rightmost $c\log n+1$ children of z or there is another occurrence of X $|\mathsf{gen}(A_m)|$ positions to the right of Y, contradicting our assumptions on Y. ◁

We have to treat a final case.

▷ Claim 18. If $r_m > 1$, $t_m = 0$ and $\mathsf{gen}(\mathsf{d}_{\mathsf{up}}(X))$ is not a suffix of $A_m^{c\log n+r_m}$ then there exists a node $v \in \mathsf{PT}(T)$ such that:

1. $\mathsf{gen}(A_0^{r_0}A_1^{r_1}\cdots A_{m-1}^{r_{m-1}}A_m)$ is a suffix of $\mathsf{Y}_\ell(v)$, and
2. $\mathsf{gen}(A_m)^{r_m-1}\mathsf{gen}(\mathsf{d}_{\mathsf{down}}(X))$ is a prefix of $\mathsf{Y}_\mathsf{r}(v)$.

Proof. In any occurrence of X in T, the label of the parent z of the rightmost node of $\mathsf{C}_{\mathsf{up}}(Y)$ is a power symbol. Let u be the r_m-th rightmost node of $\mathsf{C}_{\mathsf{up}}(Y)$. By the assumption that $\mathsf{gen}(\mathsf{d}_{\mathsf{up}}(X))$ is not a suffix of $A_m^{c\log n+r_m}$ and Lemma 13, u must be one of the $c\log n$ leftmost children of z. ◁

The combination of the two claims applied to both S and T yields the lemma. ◀

4.3 A Problem on Dynamic Bicolored Trees

Due to Lemmas 14 and 16, our task reduces to solving the problem defined below in polylogarithmic time per update, as we can directly apply it to $\mathcal{R} = \{(\mathsf{Y}_\ell(u)^R, \mathsf{Y}_\mathsf{r}(u)) : u \in \mathsf{PT}(S)\}$ and $\mathcal{B} = \{(\mathsf{Y}_\ell(v)^R, \mathsf{Y}_\mathsf{r}(v)) : v \in \mathsf{PT}(T)\}$. Note that $|\mathcal{R}| + |\mathcal{B}| = \tilde{\mathcal{O}}(n)$ throughout the execution of our algorithm.

Problem: LCP for Two Families of Pairs of Strings
Input: Two families $\mathcal{R}$ and $\mathcal{B}$, each consisting of pairs of strings, where each string is given as a handle from a dynamic collection.
Update: Insertion or deletion of an element in $\mathcal{R}$ or $\mathcal{B}$.
Query: Return $(P, Q) \in \mathcal{R}$ and $(P', Q') \in \mathcal{B}$ that maximize $\mathsf{lcp}(P, P') + \mathsf{lcp}(Q, Q')$.

Each element of $\mathcal{B}$ and $\mathcal{R}$ is given a unique identifier. We maintain two compacted tries $\mathcal{T}_P$ and $\mathcal{T}_Q$. By appending unique letters, we can assume that no string is a prefix of another string. $\mathcal{T}_P$ (resp. $\mathcal{T}_Q$) stores the string P (resp. Q) for every $(P, Q) \in \mathcal{R}$, with the corresponding leaf colored red and labeled by the identifier of the pair and the string P' (resp. Q') for every $(P', Q') \in \mathcal{B}$, with the corresponding leaf colored blue and labeled by the

identifier of the pair. Then, the sought result corresponds to a pair of nodes $u \in \mathcal{T}_P$ and $v \in \mathcal{T}_Q$ returned by a query to a data structure for the Dynamic Bicolored Trees Problem defined below for $\mathcal{T}_1 = \mathcal{T}_P$ and $\mathcal{T}_2 = \mathcal{T}_Q$, with node weights being their string-depths.

Problem: Dynamic Bicolored Trees Problem
Input: Two weighted trees $\mathcal{T}_1$ and $\mathcal{T}_2$ of total size at most m, whose leaves are bicolored and labeled, so that each label corresponds to exactly one leaf of each tree.
Update: Split an edge into two / attach a new leaf to a node / delete a leaf.
Query: Return a pair of nodes $u \in \mathcal{T}_1$ and $v \in \mathcal{T}_2$ with the maximum combined weight that have at least one red descendant with the same label, and at least one blue descendant with the same label.

To complete the reduction, we have to show how to translate an update in $\mathcal{R}$ or $\mathcal{B}$ into updates in $\mathcal{T}_P$ and $\mathcal{T}_Q$. Let us first explain how to represent $\mathcal{T}_P$ and $\mathcal{T}_Q$. For each edge, we store a handle to a string from the dynamic collection, and indices for a fragment of this string which represents the edge's label. For each explicit node, we store edges leading to its children in a dictionary structure indexed by the first letters of the edges' labels. For every leaf, we store its label and color. An insert operation receives a string (given as a handle from a dynamic collection), together with its label and color, and should create its corresponding leaf. A delete operation does not actually remove a leaf, but simply removes its label. However, in order to not increase the space complexity, we rebuild the whole data structure from scratch after every m updates. This rebuilding does not incur any extra cost asymptotically; the time required for it can be deamortized using standard techniques.

▶ **Lemma 19.** *Each update in $\mathcal{R}$ or $\mathcal{B}$ implies $\mathcal{O}(1)$ updates in $\mathcal{T}_P$ and $\mathcal{T}_Q$ that can be computed in $\mathcal{O}(\log n)$ time.*

Proof. Inserting a new leaf, corresponding to string U, to $\mathcal{T}_P$ requires possibly splitting an edge into two by creating a new explicit node, and then attaching a new leaf to an explicit node. To implement this efficiently, we maintain the set C of path-labels of explicit nodes of $\mathcal{T}_P$ in a balanced search tree, sorted in lexicographic order. Using lcp queries (cf. Theorem 12), we binary search for the longest prefix U' of U that equals the path-label of some implicit or explicit node of $\mathcal{T}_P$. If this node is explicit, then we attach a leaf to it. Otherwise, let the successor of U' in C be the path-label of node v. We split the edge $(\mathsf{parent}(v), v)$ appropriately and attach a leaf to the newly created node. This allows us to maintain $\mathcal{T}_P$ after each insert operation in $\mathcal{O}(\log n)$ time.

For a delete operation, we can access the leaf corresponding to the deleted string in $\mathcal{O}(\log n)$ time using the balanced search tree. ◀

It thus suffices to show a solution for the Dynamic Bicolored Trees Problem that processes each update in polylogarithmic time.

We will maintain a heavy-light decomposition of both $\mathcal{T}_1$ and $\mathcal{T}_2$. This can be done by using a standard method of rebuilding as used by Gabow [13]. Let $L(u)$ be the number of leaves in the subtree of u, including the leaves without labels, when the subtree was last rebuilt. Each internal node u of a tree selects at most one child v and the edge (u, v) is *heavy*. All other edges are *light*. Maximal sequences of consecutive heavy edges are called *heavy paths*. The node $r(p)$ closest to the root of the tree is called the *root* of the heavy path p and the node $e(p)$ furthest from the root of the tree is called the *end* of the heavy path. The following procedure receives a node u of the tree and recursively rebuilds its subtree.

```
1: function DECOMPOSE(u, r)                ▷ r is the root of the heavy path containing u.
2:     S ← children(u)
3:     v ← argmax_{v∈S} L(v)
4:     if L(v) ≥ 5/6 · L(u) then
5:         edge (u, v) is heavy
6:         DECOMPOSE(v, r)
7:         S ← S \ {v}
8:     for v ∈ S do
9:         DECOMPOSE(v, v)
```

Every root u of a heavy path maintains the number of insertions $I(u)$ in its subtree since it was last rebuilt. When $I(u) \geq \frac{1}{6} \cdot L(u)$, we recalculate the values of $L(v)$ for nodes v in the subtree of u and call DECOMPOSE(u, u). This maintains the property that $L(e(p)) \geq \frac{2}{3} L(r(p))$ for each heavy path p and leads to the following.

▶ **Proposition 20.** *There are $\mathcal{O}(\log m)$ heavy paths above any node.*

As rebuilding a subtree of size s takes $\mathcal{O}(s)$ time, by a standard potential argument, we get the following.

▶ **Lemma 21.** *The heavy-light decompositions of $\mathcal{T}_1$ and $\mathcal{T}_2$ can be maintained in $\mathcal{O}(\log m)$ amortized time per update.*

The main ingredient of our structure is a collection of additional structures, each storing a dynamic set of points. Each such point structure sends its current result to a max-heap, and after each update we return the largest element stored in the heap. The problem each of these point structures are designed for is the following.

Problem: Dynamic Best Bichromatic Point
Input: A multiset of at most m bicolored points from $[m] \times [m]$.
Update: Insertions and deletions of points from $[m] \times [m]$.
Query: Return a pair of points $R = (x, y)$ and $B = (x', y')$ such that R is red, B is blue, and $\min(x, x') + \min(y, y')$ is as large as possible.

We call the pair of points sought in this problem the *best bichromatic pair of points*. In Section 4.4 we explain how to modify range trees in order to obtain the following result.

▶ **Lemma 22.** *There is a data structure for Dynamic Best Bichromatic Point that processes each update in $\mathcal{O}(\log^2 m)$ amortized time.*

Conceptually, we maintain a point structure for every pair of heavy paths from $\mathcal{T}_P$ and $\mathcal{T}_Q$. However, the total number of points stored in all structures at any moment is only $\mathcal{O}(m \log^2 m)$ and the empty structures are not actually created. Consider heavy paths p of $\mathcal{T}_1$ and q of $\mathcal{T}_2$. Let ℓ be a label such that there are leaves u in the subtree of $r(p)$ in $\mathcal{T}_1$ and v in the subtree of $r(q)$ in $\mathcal{T}_2$ with the same color and both labeled by ℓ. Then, the point structure should contain a point (x, y) with this color, where x and y are the string-depths of the nodes of p and q containing u and v in their light subtrees, respectively. It can be verified that then the answer extracted from the point structure is equal to the sought result, assuming that the corresponding pair of nodes belongs to p and q, respectively. It remains to explain how to maintain this invariant when both trees undergo modifications.

Splitting an edge does not require any changes to the point structures. Each label appears only once in $\mathcal{T}_1$ and $\mathcal{T}_2$, and hence by Proposition 20 contributes to only $\mathcal{O}(\log^2 m)$ point structures. Furthermore, by navigating the heavy path decompositions we can access these structures efficiently. This allows us to implement each deletion in $\mathcal{O}(\log^4 m)$ amortized time, employing Lemma 22. To implement the insertions, we need to additionally explain what to do after rebuilding a subtree of u. In this case, we first remove all points corresponding to leaves in the subtree of u, then rebuild the subtree, and then proceed to insert points to existing and potentially new point structures. This can be amortized by the same standard potential argument if we add another factor of $\mathcal{O}(\log^2 n)$ in the analysis to account for the fact that we add a point in $\mathcal{O}(\log^2 n)$ point structures for each leaf in the subtree of u. Thus, insertions require $\mathcal{O}(\log^5 n)$ amortized time as well.

Wrap-up. Lemma 16 reduces our problem to the LCP for Two Families of Pairs of Strings problem for sets $\mathcal{R}$ and $\mathcal{B}$ of size $\tilde{\mathcal{O}}(n)$, so that each substitution in S or T yields $\tilde{\mathcal{O}}(1)$ updates to $\mathcal{R}$ and $\mathcal{B}$, which can be computed in $\tilde{\mathcal{O}}(1)$ time due to Lemma 14. The LCP for Two Families of Pairs of Strings problem is then reduced to the Dynamic Bicolored Trees Problem for trees $\mathcal{T}_1$ and $\mathcal{T}_2$ of size $\tilde{\mathcal{O}}(n)$, so that each update in $\mathcal{R}$ or $\mathcal{B}$ yields $\mathcal{O}(1)$ updates to the trees, which can be computed in $\mathcal{O}(\log n)$ time (Lemma 19). We solve the latter problem by maintaining a heavy-light decomposition of each of the trees in $\mathcal{O}(\log n)$ amortized time per update (Lemma 21), and an instance of a data structure for the Dynamic Best Bichromatic Point problem for each pair of heavy paths. For each update to the trees, we spend $\mathcal{O}(\log^5 n)$ amortized time to update the point structures.

4.4 Dynamic Best Bichromatic Point

In this section we prove Lemma 22, i.e. design an efficient data structure for the Dynamic Best Bichromatic Point problem.

With standard perturbation, we can guarantee that all x and y coordinates of points are distinct. We maintain an augmented dynamic 2D range tree [31] over the multiset of points. This is a balanced search tree $\mathcal{T}$ (called primary) over the x coordinates of all points in the multiset in which every x coordinate corresponds to a leaf and, more generally, every node $u \in \mathcal{T}$ corresponds to a range of x coordinates denoted by $x(u)$. Additionally, every $u \in \mathcal{T}$ stores another balanced search tree $\mathcal{T}_u$ (called secondary) over the y coordinates of all points $(x, y) \in S$ such that $x \in x(u)$. Thus, the leaves of $\mathcal{T}_u$ correspond to y coordinates of such points, and every $v \in \mathcal{T}_u$ corresponds to a range of y coordinates denoted by $y(v)$. We interpret every $v \in \mathcal{T}_u$ as the rectangular region of the plane $x(u) \times y(v)$, and, in particular, each leaf $v \in \mathcal{T}_u$ corresponds to a single point in the multiset. Each node $v \in \mathcal{T}_u$ will be augmented with some extra information that can be computed in constant time from the extra information stored in its children. Similarly, each node $u \in \mathcal{T}$ will be augmented with some extra information that can be computed in constant time from the extra information stored in its children together with the extra information stored in the root of the secondary tree $\mathcal{T}_u$. Irrespectively of what this extra information is, as explained by Willard and Lueker [31], if we implement the primary tree as a $BB(\alpha)$ tree and each secondary tree as a balanced search tree, each insertion and deletion can be implemented in $\mathcal{O}(\log^2 m)$ amortized time.

Before we explain what is the extra information, we need the following notion. Consider a non-leaf node $u \in \mathcal{T}$ and let $u_\ell, u_r \in \mathcal{T}$ be its children. Let $v \in \mathcal{T}_u$ be a non-leaf node with children $v_\ell, v_r \in \mathcal{T}_u$. The regions $A = x(u_\ell) \times y(v_\ell)$, $B = x(u_\ell) \times y(v_r)$, $C = x(u_r) \times y(v_\ell)$ and $D = x(u_r) \times y(v_r)$ partition $x(u) \times y(v)$ into four parts. We say that two points $p = (x, y)$

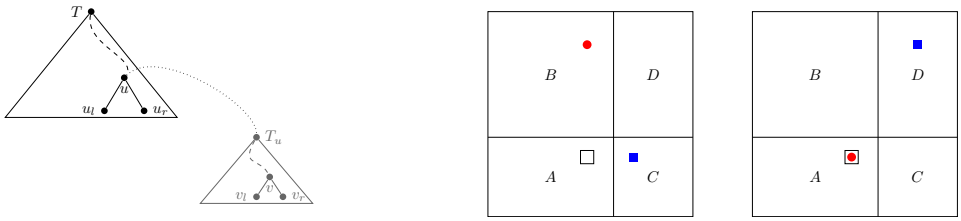

Figure 2 Left: A 2D range tree. Right: Node representing regions A, B, C, D. The best pair for each case is denoted by a small square.

and $q = (x', y')$ with $x < x'$ are shattered by $v \in \mathcal{T}_u$ if and only if $p \in A$ and $q \in D$ or $p \in B$ and $q \in C$ (note that the former is only possible when $y < y'$ while the latter can only hold when $y > y'$).

▶ **Proposition 23.** *Any pair of points in the multiset is shattered by a unique $v \in \mathcal{T}_u$ (for a unique u).*

Now we are ready to describe the extra information. Each node $u \in \mathcal{T}$ stores the best bichromatic pair with x coordinates from $x(u)$. Each node $v \in \mathcal{T}_u$ stores the best bichromatic pair shattered by one of its descendants $v' \in \mathcal{T}_u$ (possibly v itself). Additionally, each node $v \in \mathcal{T}_u$ stores the following information about points of each color in its region:

1. the point with the maximum x,
2. the point with the maximum y,
3. a point with the maximum $x + y$.

We need to verify that such extra information can be indeed computed in constant time from the extra information stored in the children.

▶ **Lemma 24.** *Let $v \in \mathcal{T}_u$ be a non-leaf node, and v_ℓ, v_r be its children. The extra information of v can be computed in constant time given the extra information stored in v_ℓ and v_r.*

Proof. This is clear for the maximum x, y and $x + y$ of each color, as we can take the maximum of the corresponding values stored in the children. For the best bichromatic pair shattered by a descendant v' of v, we start with considering the best bichromatic pair shattered by a descendant v'_ℓ of v_ℓ and v'_r of v_r. The remaining case is that the best bichromatic pair is shattered by v itself. Let A, B, C, D be as in the definition of shattering. Without losing generality we assume that the sought pair is $p = (x, y)$ and $q = (x', y')$ with $x < x'$, red p and blue q. We consider two cases:

1. $p \in A$ and $q \in D$: the best such pair is obtained by taking p with the maximum $x + y$ and any q,
2. $p \in B$ and $q \in C$: the best such pair is obtained by taking p with the maximum x and q with the maximum y.

In both cases, we are able to compute the best bichromatic pair shattered by v using the extra information stored at the children of v. See Figure 2. ◀

▶ **Lemma 25.** *Let $u \in \mathcal{T}$ be a non-leaf node, and u_ℓ, u_r be its children. The extra information of v can be computed in constant time given the extra information stored in v_ℓ, v_r and the root of $\mathcal{T}_u$.*

Proof. We seek the best bichromatic pair with x coordinates from $x(u)$. If the x coordinates are in fact from $x(u_\ell)$ or $x(u_r)$, we obtain the pair from the children of u. Otherwise, the pair must be shattered by some $v \in \mathcal{T}_u$ that is a descendant of the root of $\mathcal{T}_u$, so we obtain the pair from the root of $\mathcal{T}_u$. ◀

References

1 Paniz Abedin, Sahar Hooshmand, Arnab Ganguly, and Sharma V. Thankachan. The heaviest induced ancestors problem revisited. In *29th CPM*, pages 20:1–20:13, 2018. doi:10.4230/LIPIcs.CPM.2018.20.

2 Stephen Alstrup, Gerth Stølting Brodal, and Theis Rauhe. Pattern matching in dynamic texts. In *11th SODA*, pages 819–828, 2000. URL: http://dl.acm.org/citation.cfm?id=338219.338645.

3 Amihood Amir and Itai Boneh. Locally maximal common factors as a tool for efficient dynamic string algorithms. In *29th CPM*, pages 11:1–11:13, 2018. doi:10.4230/LIPIcs.CPM.2018.11.

4 Amihood Amir and Itai Boneh. Dynamic palindrome detection. *CoRR*, abs/1906.09732, 2019. arXiv:1906.09732.

5 Amihood Amir, Itai Boneh, Panagiotis Charalampopoulos, and Eitan Kondratovsky. Repetition Detection in a Dynamic String. In *27th ESA*, pages 5:1–5:18, 2019. doi:10.4230/LIPIcs.ESA.2019.5.

6 Amihood Amir, Panagiotis Charalampopoulos, Costas S. Iliopoulos, Solon P. Pissis, and Jakub Radoszewski. Longest common factor after one edit operation. In *24th SPIRE*, pages 14–26, 2017. doi:10.1007/978-3-319-67428-5_2.

7 Amihood Amir, Panagiotis Charalampopoulos, Solon P. Pissis, and Jakub Radoszewski. Longest common substring made fully dynamic. In *27th ESA*, pages 6:1–6:17, 2019. doi:10.4230/LIPIcs.ESA.2019.6.

8 Amihood Amir, Gad M. Landau, Moshe Lewenstein, and Dina Sokol. Dynamic text and static pattern matching. *ACM Trans. Algorithms*, 3(2):19, 2007. doi:10.1145/1240233.1240242.

9 Panagiotis Charalampopoulos, Maxime Crochemore, Costas S. Iliopoulos, Tomasz Kociumaka, Solon P. Pissis, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. Linear-time algorithm for long LCF with k mismatches. In *29th CPM*, pages 23:1–23:16, 2018. doi:10.4230/LIPIcs.CPM.2018.23.

10 Richard Cole, Lee-Ad Gottlieb, and Moshe Lewenstein. Dictionary matching and indexing with errors and don't cares. In *36th STOC*, pages 91–100, 2004. doi:10.1145/1007352.1007374.

11 Martin Farach. Optimal suffix tree construction with large alphabets. In *38th FOCS*, pages 137–143, 1997. doi:10.1109/SFCS.1997.646102.

12 Martin Farach and S. Muthukrishnan. Perfect hashing for strings: Formalization and algorithms. In *7th CPM*, pages 130–140, 1996. doi:10.1007/3-540-61258-0_11.

13 Harold N. Gabow. Data structures for weighted matching and nearest common ancestors with linking. In *1st SODA*, pages 434–443, 1990. URL: http://dl.acm.org/citation.cfm?id=320176.320229.

14 Travis Gagie, Paweł Gawrychowski, and Yakov Nekrich. Heaviest induced ancestors and longest common substrings. In *25th CCCG*, 2013. URL: http://cccg.ca/proceedings/2013/papers/paper_29.pdf.

15 Paweł Gawrychowski, Adam Karczmarz, Tomasz Kociumaka, Jakub Lacki, and Piotr Sankowski. Optimal dynamic strings. In *29th SODA*, pages 1509–1528, 2018. doi:10.1137/1.9781611975031.99.

16 Monika Henzinger, Sebastian Krinninger, Danupon Nanongkai, and Thatchaphol Saranurak. Unifying and strengthening hardness for dynamic problems via the online matrix-vector multiplication conjecture. In *47th STOC*, pages 21–30, 2015. doi:10.1145/2746539.2746609.

17 Tomohiro I. Longest common extensions with recompression. In *28th CPM*, pages 18:1–18:15, 2017. doi:10.4230/LIPIcs.CPM.2017.18.

18 Artur Jeż. Faster fully compressed pattern matching by recompression. *ACM Transactions on Algorithms*, 11(3):20:1–20:43, 2015. doi:10.1145/2631920.

19 Artur Jeż. Recompression: A simple and powerful technique for word equations. *J. ACM*, 63(1):4:1–4:51, 2016. doi:10.1145/2743014.

20 Tomasz Kociumaka, Jakub Radoszewski, and Tatiana A. Starikovskaya. Longest common substring with approximately k mismatches. *Algorithmica*, 81(6):2633–2652, 2019. doi:10.1007/s00453-019-00548-x.

21 Tomasz Kociumaka, Tatiana A. Starikovskaya, and Hjalte Wedel Vildhøj. Sublinear space algorithms for the longest common substring problem. In *22nd ESA*, pages 605–617, 2014. doi:10.1007/978-3-662-44777-2_50.

22 Yakov Nekrich. A data structure for multi-dimensional range reporting. In *23rd SOCG*, pages 344–353, 2007. doi:10.1145/1247069.1247130.

23 Mihai Patrascu. Unifying the landscape of cell-probe lower bounds. *SIAM J. Comput.*, 40(3):827–847, 2011. doi:10.1137/09075336X.

24 Mihai Pătraşcu and Erik D. Demaine. Logarithmic lower bounds in the cell-probe model. *SIAM J. Comput.*, 35(4):932–963, 2006. doi:10.1137/S0097539705447256.

25 Mihai Pătraşcu and Mikkel Thorup. Time-space trade-offs for predecessor search. In *38th STOC*, pages 232–240, 2006. doi:10.1145/1132516.1132551.

26 S. Sahinalp and U. Vishkin. Efficient approximate and dynamic matching of patterns using a labeling paradigm. In *54th FOCS*, pages 320–328, 1996. doi:10.1109/SFCS.1996.548491.

27 Tatiana A. Starikovskaya and Hjalte Wedel Vildhøj. Time-space trade-offs for the longest common substring problem. In *24th CPM*, pages 223–234, 2013. doi:10.1007/978-3-642-38905-4_22.

28 Sharma V. Thankachan, Alberto Apostolico, and Srinivas Aluru. A provably efficient algorithm for the k-mismatch average common substring problem. *Journal of Computational Biology*, 23(6):472–482, 2016. doi:10.1089/cmb.2015.0235.

29 Peter Weiner. Linear pattern matching algorithms. In *14th FOCS*, pages 1–11, 1973. doi:10.1109/SWAT.1973.13.

30 Dan E. Willard. Log-logarithmic worst-case range queries are possible in space $\Theta(n)$. *Information Processing Letters*, 17(2):81–84, 1983. doi:10.1016/0020-0190(83)90075-3.

31 Dan E. Willard and George S. Lueker. Adding range restriction capability to dynamic data structures. *J. ACM*, 32(3):597–617, 1985. doi:10.1145/3828.3839.