Faster Algorithms for Longest Common Substring

Panagiotis Charalampopoulos, Tomasz Kociumaka, Solon P Pissis, Jakub Radoszewski

Faster Algorithms for Longest Common Substring

Panagiotis Charalampopoulos
The Interdisciplinary Center Herzliya, Israel

Tomasz Kociumaka
University of California, Berkeley, USA

Solon P. Pissis
CWI, Amsterdam, The Netherlands
Vrije Universiteit, Amsterdam, The Netherlands

Jakub Radoszewski
Institute of Informatics, University of Warsaw, Poland
Samsung R&D Poland

Abstract

In the classic longest common substring (LCS) problem, we are given two strings S and T, each of length at most n, over an alphabet of size σ, and we are asked to find a longest string occurring as a fragment of both S and T. Weiner, in his seminal paper that introduced the suffix tree, presented an $\mathcal{O}(n \log \sigma)$-time algorithm for this problem [SWAT 1973]. For polynomially-bounded integer alphabets, the linear-time construction of suffix trees by Farach yielded an $\mathcal{O}(n)$-time algorithm for the LCS problem [FOCS 1997]. However, for small alphabets, this is not necessarily optimal for the LCS problem in the word RAM model of computation, in which the strings can be stored in $\mathcal{O}(n \log \sigma / \log n)$ space and read in $\mathcal{O}(n \log \sigma / \log n)$ time. We show that, in this model, we can compute an LCS in time $\mathcal{O}(n \log \sigma / \sqrt{\log n})$, which is sublinear in n if $\sigma = 2^{o(\sqrt{\log n})}$ (in particular, if $\sigma = \mathcal{O}(1)$), using optimal space $\mathcal{O}(n \log \sigma / \log n)$.

We then lift our ideas to the problem of computing a k-mismatch LCS, which has received considerable attention in recent years. In this problem, the aim is to compute a longest substring of S that occurs in T with at most k mismatches. Flouri et al. showed how to compute a 1-mismatch LCS in $\mathcal{O}(n \log n)$ time [IPL 2015]. Thankachan et al. extended this result to computing a k-mismatch LCS in $\mathcal{O}(n \log^k n)$ time for $k = \mathcal{O}(1)$ [J. Comput. Biol. 2016]. We show an $\mathcal{O}(n \log^{k-1/2} n)$-time algorithm, for any constant $k > 0$ and *irrespective* of the alphabet size, using $\mathcal{O}(n)$ space as the previous approaches. We thus notably break through the well-known $n \log^k n$ barrier, which stems from a recursive heavy-path decomposition technique that was first introduced in the seminal paper of Cole et al. [STOC 2004] for string indexing with k errors.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases longest common substring, k mismatches, wavelet tree

Funding *Panagiotis Charalampopoulos*: Supported by the Israel Science Foundation grant 592/17.
Tomasz Kociumaka: Partly supported by NSF 1652303, 1909046, and HDR TRIPODS 1934846 grants, and an Alfred P. Sloan Fellowship.
Jakub Radoszewski: Supported by the Polish National Science Center, grant no. 2018/31/D/ST6/03991.

1 Introduction

In the classic longest common substring (LCS) problem, we are given two strings S and T, each of length at most n, over an alphabet of size σ, and we are asked to find a longest string occurring as a fragment of both S and T. The problem was conjectured by Knuth to require $\Omega(n \log n)$ time until Weiner, in his paper introducing the suffix tree [66], showed that the LCS problem can be solved in $\mathcal{O}(n)$ time when σ is constant via constructing the suffix tree of string $S\#T$, for a sentinel letter $\#$. Later, Farach showed that if σ is not constant, the suffix tree can be constructed in linear time in addition to the time required for sorting its letters [35]. This yielded an $\mathcal{O}(n)$-time algorithm for the LCS problem in the word RAM model for polynomially-bounded integer alphabets. While Farach's algorithm for suffix tree construction is *optimal* for all alphabets (the suffix tree by definition has size $\Theta(n)$), the same does not hold for the LCS problem. We were thus motivated to answer the following basic question:

Can the LCS problem be solved in $o(n)$ time when $\log\sigma = o(\log n)$?

We consider the word RAM model and assume an alphabet $[0, \sigma)$. Any string of length n can then be stored in $\mathcal{O}(n\log\sigma/\log n)$ space and read in $\mathcal{O}(n\log\sigma/\log n)$ time. Note that if $\log\sigma = \Theta(\log n)$, one requires $\Theta(n)$ time to read the input. We answer this basic question positively when $\log\sigma = o(\sqrt{\log n})$:

▶ **Theorem 1.** *Given two strings S and T, each of length at most n, over an alphabet $[0, \sigma)$, we can solve the LCS problem in $\mathcal{O}(n\log\sigma/\sqrt{\log n})$ time using $\mathcal{O}(n/\log_\sigma n)$ space.*

We also consider the following generalisation of the LCS problem that allows mismatches.

k-MISMATCH LONGEST COMMON SUBSTRING (k-LCS) **Input:** Two strings S, T of length up to n over an integer alphabet and an integer k. **Output:** A pair S', T' of substrings of S and T, respectively, with Hamming distance (i.e., number of mismatches) at most k and maximal length.

Flouri et al. presented an $\mathcal{O}(n\log n)$-time algorithm for the 1-LCS problem [37]. (Earlier work on this problem includes [6].) This was generalised by Thankachan et al. [63] to an algorithm for the k-LCS problem that works in $\mathcal{O}(n\log^k n)$ time if $k = \mathcal{O}(1)$. Both algorithms use $\mathcal{O}(n)$ space. In [26], Charalampopoulos et al. presented an $\mathcal{O}(n + n\log^{k+1} n/\sqrt{\ell})$-time algorithm for k-LCS with $k = \mathcal{O}(1)$, where ℓ is the length of a k-LCS. For general k, Flouri et al. presented an $\mathcal{O}(n^2)$-time algorithm that uses $\mathcal{O}(1)$ additional space [37]. Grabowski [43] presented two algorithms with running times $\mathcal{O}(n((k+1)(\ell_0+1))^k)$ and $\mathcal{O}(n^2k/\ell_k)$, where ℓ_0 and ℓ_k are, respectively, the length of an LCS of S and T and the length of a k-LCS of S and T. Abboud et al. [1] employed the polynomial method to obtain a $k^{1.5}n^2/2^{\Omega(\sqrt{\log n/k})}$-time randomised algorithm. In [52], Kociumaka et al. showed that, assuming the Strong Exponential Time Hypothesis (SETH) [48, 49], no strongly subquadratic-time solution for k-LCS exists for $k = \Omega(\log n)$. The authors of [52] additionally presented a subquadratic-time 2-approximation algorithm for k-LCS for general k.

Analogously to Weiner's solution to the LCS problem via suffix trees, the algorithm of Thankachan et al. [63] builds upon the ideas of the k-errata tree, which was introduced by Cole et al. [32] in their seminal paper for indexing a string of length n with the aim of answering pattern matching queries with up to k mismatches. For constant k, the size of the k-errata tree is $\mathcal{O}(n\log^k n)$. (Note that computing a k-LCS using the k-errata tree directly is not straightforward as opposed to computing an LCS using the suffix tree.)

We show the following result, breaking through the $n\log^k n$ barrier, for any constant integer $k > 0$ and irrespective of the alphabet size. Note that, in the word RAM, the letters of S and T can be renumbered in $\mathcal{O}(n\log\log n)$ time [45] so that they belong to $[0, \sigma)$.

▶ **Theorem 2.** *Given two strings S and T, each of length at most n, and a constant integer $k > 0$, the k-LCS problem can be solved in $\mathcal{O}(n\log^{k-1/2} n)$ time using $\mathcal{O}(n)$ space.*

Notably, on the way to proving the above theorem, we improve upon [26] by showing an $\mathcal{O}(n + n\log^{k+1} n/\ell)$-time algorithm for k-LCS with $k = \mathcal{O}(1)$, where ℓ is the length of a k-LCS. (Our second summand is smaller by a $\sqrt{\ell}$ multiplicative factor compared to [26].)

Our Techniques

At the heart of our approaches lies the following Two String Families LCP Problem. (Here, the length of the longest common prefix of two strings U and V is denoted by $\mathsf{LCP}(U, V)$; see Preliminaries for a precise definition of compacted tries.)

> Two String Families LCP Problem
> **Input:** Compacted tries $\mathcal{T}(\mathcal{F}_1), \mathcal{T}(\mathcal{F}_2)$ of $\mathcal{F}_1, \mathcal{F}_2 \subseteq \Sigma^*$ and two sets $\mathcal{P}, \mathcal{Q} \subseteq \mathcal{F}_1 \times \mathcal{F}_2$, with $|\mathcal{P}|, |\mathcal{Q}|, |\mathcal{F}_1|, |\mathcal{F}_2| \leq N$.
> **Output:** The value
> $\mathrm{maxPairLCP}(\mathcal{P}, \mathcal{Q}) = \max\{\mathsf{LCP}(P_1, Q_1) + \mathsf{LCP}(P_2, Q_2) : (P_1, P_2) \in \mathcal{P}, (Q_1, Q_2) \in \mathcal{Q}\}$.

This abstract problem was introduced in [26]. Its solution, shown in the lemma below, is directly based on a technique that was used in [20, 34] and then in [37] to devise an $\mathcal{O}(n \log n)$-time solution for 1-LCS. In particular, Lemma 3 immediately implies an $\mathcal{O}(n \log n)$-time algorithm for 1-LCS.

▶ **Lemma 3** ([26, Lemma 3]). *The Two String Families LCP Problem can be solved in $\mathcal{O}(N \log N)$ time and $\mathcal{O}(N)$ space.*[1]

In the algorithm underlying Lemma 3, for each node v of $\mathcal{T}(\mathcal{F}_1)$ we try to identify a pair of elements, one from $\mathcal{P}$ and one from $\mathcal{Q}$, whose first components are descendants of v and the LCP of their second components is maximised. The algorithm traverses $\mathcal{T}(\mathcal{F}_1)$ bottom-up and uses mergeable height-balanced trees with $\mathcal{O}(N \log N)$ total merging time to store elements of pairs; see [21].

An $o(N \log N)$ time solution to the Two String Families LCP Problem is not known and devising such an algorithm seems hard. The key ingredient of our algorithms is an efficient solution to the following special case of the problem. We say that a family of string pairs $\mathcal{P}$ is an (α, β)-*family* if each $(U, V) \in \mathcal{P}$ satisfies $|U| \leq \alpha$ and $|V| \leq \beta$.

▶ **Lemma 4.** *An instance of the Two String Families LCP Problem in which $\mathcal{P}$ and $\mathcal{Q}$ are (α, β)-families can be solved in time $\mathcal{O}(N(\alpha + \log N)(\log \beta + \sqrt{\log N})/\log N)$ and space $\mathcal{O}(N + N\alpha/\log N)$.*

The algorithm behind this solution uses a wavelet tree of the first components of $\mathcal{P} \cup \mathcal{Q}$.

Solution to LCS. For the LCS problem, we design three different algorithms depending on the length of the solution. For short LCS ($\leq \frac{1}{3} \log_\sigma n$), we employ a simple tabulation technique. For long LCS ($\geq \log^4 n$), we employ the technique of Charalampopoulos et al. [26] for computing a long k-LCS, plugging in the sublinear LCE data structure of Kempa and Kociumaka [50]. Both of these solutions work in $\mathcal{O}(n/\log_\sigma n)$ time.

As for medium-length LCS, let us first consider a case when the strings do not contain highly periodic fragments. In this case, we use the string synchronising sets of Kempa and Kociumaka [50] to select a set of $\mathcal{O}(\frac{n}{\tau})$ anchors over S and T, where $\tau = \Theta(\log_\sigma n)$, such that for any common substring U of S and T of length $\ell \geq 3\tau - 1$, there exist occurrences $S[i^S \mathinner{.\,.} j^S]$ and $T[i^T \mathinner{.\,.} j^T]$ of U, for which we have anchors $a^S \in [i^S, j^S]$ and $a^T \in [i^T, j^T]$ with $a^S - i^S = a^T - i^T \leq \tau$. For each anchor a in S, we add a string pair $((S[a - \tau \mathinner{.\,.} a))^R, S[a \mathinner{.\,.} a + \beta))$ to $\mathcal{P}$ (and similarly for T and $\mathcal{Q}$). This lets us apply Lemma 4

[1] The original formulation of [26, Lemma 3] does not discuss the space complexity. However, it can be readily verified that the underlying algorithm, described in [34, 37], uses linear space.

with $N = \mathcal{O}(n/\tau)$, $\alpha = \mathcal{O}(\tau)$, and $\beta = \mathcal{O}(\log^4 n)$. In the periodic case, we cannot guarantee that $a^S - i^S = a^T - i^T$ is small, but we can obtain a different set of anchors based on maximal repetitions (runs) that yields multiple instances of the Two String Families LCP Problem, which have extra structure leading to a linear-time solution.

Solution to k-LCS. In this case we also obtain a set of $\mathcal{O}(n/\ell)$ anchors, where ℓ is the length of k-LCS. If the common substring is far from highly periodic, we use a synchronising set for $\tau = \Theta(\ell)$, and otherwise we generate anchors using a technique of misperiods that was initially introduced for k-mismatch pattern matching [19, 29]. Now the families $\mathcal{P}, \mathcal{Q}$ need to consist not simply of substrings of S and T, but rather of modified substrings generated by an approach that resembles k-errata trees [32]. This requires combining the ideas of Thankachan et al. [63] and Charalampopoulos et al. [26]; this turns out to be technically challenging in order to stay within linear space. We apply Lemma 3 or Lemma 4 depending on the length ℓ, which allows us to break through the $n \log^k n$ barrier for k-LCS.

Other Related Work

A large body of work has been devoted to exploiting bit-parallelism in the word RAM model for string matching [7, 61, 59, 38, 39, 51, 15, 23, 9, 16, 11, 18, 42, 12, 17].

Other results on the LCS problem include the linear-time computation of an LCS of several strings over an integer alphabet [46], trade-offs between the time and the working space for computing an LCS of two strings [13, 53, 60], and the dynamic maintenance of an LCS [2, 3, 27]. Very recently, a strongly sublinear-time quantum algorithm and a lower bound for the quantum setting were shown [41]. The k-LCS problem has also been studied under edit distance and subquadratic-time algorithms for $k = o(\log n)$ are known [62, 4].

The problem of indexing a string of length n over an alphabet $[0, \sigma)$ in sublinear time in the word RAM model, with the aim of answering pattern matching queries, has attracted significant attention. Since by definition the suffix tree occupies $\Theta(n)$ space, alternative indexes have been sought. The state of the art is an index that occupies $\mathcal{O}(n \log \sigma / \log n)$ space and can be constructed in $\mathcal{O}(n \log \sigma / \sqrt{\log n})$ time [50, 57]. Interestingly, the running time of our algorithm (Theorem 1) matches the construction time of this index. Note that, in contrast to suffix trees, such indexes *cannot be used directly* for computing an LCS. Intuitively, these indexes sample suffixes of the string to be indexed, and upon a pattern matching query, they have to treat separately the first $\mathcal{O}(\log_\sigma n)$ letters of the pattern.

As for k-mismatch indexing, for $k = \mathcal{O}(1)$, a k-errata tree occupies $\mathcal{O}(n \log^k n)$ space, can be constructed in $\mathcal{O}(n \log^{k+1} n)$ time, and supports pattern matching queries with at most k mismatches in $\mathcal{O}(m + \log^k n \log\log n + occ)$ time, where m is the length of the pattern and occ is the number of the reported pattern occurrences. Other trade-offs for this problem, in which the product of space and query time is still $\Omega(n \log^{2k} n)$, were shown in [25, 64], and solutions with $\mathcal{O}(n)$ space but $\Omega(\min\{n, \sigma^k m^{k-1}\})$-time queries were presented in [24, 30, 47, 65]. More efficient solutions for $k = 1$ are known (see [10] and references therein). Cohen-Addad et al. [31] showed that, under SETH, for $k = \Theta(\log n)$ any indexing data structure that can be constructed in polynomial time cannot have $\mathcal{O}(n^{1-\delta})$ query time, for any $\delta > 0$. They also showed that in the pointer machine model, for $k = o(\log n)$, exponential dependency on k either in the space or in the query time cannot be avoided (for the reporting version of the problem). We hope that our techniques can fuel further progress in k-mismatch indexing.

2 Preliminaries

Strings. Let $T = T[1]T[2]\cdots T[n]$ be a *string* (or *text*) of length $n = |T|$ over an alphabet $\Sigma = [0, \sigma)$. The elements of Σ are called *letters*.

By ε we denote the *empty string*. For two positions i and j of T, we denote by $T[i\,..\,j]$ the *fragment* of T that starts at position i and ends at position j (the fragment is empty if $i > j$). A fragment of T is represented using $\mathcal{O}(1)$ space by specifying the indices i and j. We define $T[i\,..\,j) = T[i\,..\,j-1]$ and $T(i\,..\,j] = T[i+1\,..\,j]$. The fragment $T[i\,..\,j]$ is an *occurrence* of the underlying *substring* $P = T[i]\cdots T[j]$. We say that P occurs at *position* i in T. A *prefix* of T is a fragment of T of the form $T[1\,..\,j]$ and a *suffix* of T is a fragment of T of the form $T[i\,..\,n]$. We denote the *reverse string* of T by T^R, i.e., $T^R = T[n]T[n-1]\cdots T[1]$. By UV we denote the *concatenation* of two strings U and V, i.e., $UV = U[1]U[2]\cdots U[|U|]V[1]V[2]\cdots V[|V|]$.

A positive integer p is called a *period* of a string T if $T[i] = T[i+p]$ for all $i \in [1, |T|-p]$. We refer to the smallest period as *the period* of the string, and denote it by $\mathsf{per}(T)$. A string T is called *periodic* if $\mathsf{per}(T) \le |T|/2$ and *aperiodic* otherwise. A *run* in T is a periodic substring that cannot be extended (to the left nor to the right) without an increase of its shortest period. All runs in a string can be computed in linear time [8, 54].

▶ **Lemma 5** (Periodicity Lemma (weak version) [36])**.** *If a string S has periods p and q such that $p + q \le |S|$, then $\gcd(p, q)$ is also a period of S.*

Tries. Let $\mathcal{M}$ be a finite set containing $m > 0$ strings over Σ. The *trie* of $\mathcal{M}$, denoted by $\mathcal{R}(\mathcal{M})$, contains a node for every distinct prefix of a string in $\mathcal{M}$; the root node is ε; the set of leaf nodes is $\mathcal{M}$; and edges are of the form $(u, \alpha, u\alpha)$, where u and $u\alpha$ are nodes and $\alpha \in \Sigma$. The *compacted trie* of $\mathcal{M}$, denoted by $\mathcal{T}(\mathcal{M})$, contains the root, the branching nodes, and the leaf nodes of $\mathcal{R}(\mathcal{M})$. Each maximal branchless path segment from $\mathcal{R}(\mathcal{M})$ is replaced by a single edge, and a fragment of a string $M \in \mathcal{M}$ is used to represent the label of this edge in $\mathcal{O}(1)$ space. The best known example of a compacted trie is the suffix tree [66]. Throughout our algorithms, $\mathcal{M}$ always consists of a set of substrings or modified substrings with $k = \mathcal{O}(1)$ modifications (see Section 5 for a definition) of a reference string. The value $\mathsf{val}(u)$ of a node u is the concatenation of labels of edges on the path from the root to u, and the *string-depth* of u is the length of $\mathsf{val}(u)$. The size of $\mathcal{T}(\mathcal{M})$ is $\mathcal{O}(m)$. We use the following well-known construction (cf. [33]).

▶ **Lemma 6.** *Given a sorted list of N strings with longest common prefixes between pairs of consecutive strings, the compacted trie of the strings can be constructed in $\mathcal{O}(N)$ time.*

Packed strings. We assume the unit-cost word RAM model with word size $w = \Theta(\log n)$ and a standard instruction set including arithmetic operations, bitwise Boolean operations, and shifts. We count the space complexity of our algorithms in machine words used by the algorithm. The *packed representation* of a string T over alphabet $[0, \sigma)$ is a list obtained by storing $\Theta(\log_\sigma n)$ letters per machine word thus representing T in $\mathcal{O}(|T|/\log_\sigma n)$ machine words. If T is given in the packed representation we simply say that T is a *packed string*.

String synchronising sets. Our solution uses the string synchronising sets introduced by Kempa and Kociumaka [50]. Informally, in the simpler case that T is cube-free, a τ-synchronising set of T is a small set of *synchronising positions* in T such that each length-τ fragment of T contains at least one synchronising position, and the leftmost synchronising positions within two sufficiently long matching fragments of T are consistent.

Formally, for a string T and a positive integer $\tau \le \frac{1}{2}n$, a set $A \subseteq [1, n-2\tau+1]$ is a *τ-synchronising set* of T if it satisfies the following two conditions:

1. If $T[i\,..\,i+2\tau) = T[j\,..\,j+2\tau)$, then $i \in A$ if and only if $j \in A$.
2. For $i \in [1, n-3\tau+2]$, $A \cap [i, i+\tau) = \emptyset$ if and only if $\mathsf{per}(T[i\,..\,i+3\tau-2]) \le \frac{1}{3}\tau$.

▶ **Theorem 7** ([50, Proposition 8.10, Theorem 8.11]). *For a string $T \in [0,\sigma)^n$ with $\sigma = n^{\mathcal{O}(1)}$ and $\tau \le \frac{1}{2}n$, there exists a τ-synchronising set of size $\mathcal{O}(n/\tau)$ that can be constructed in $\mathcal{O}(n)$ time or, if $\tau \le \frac{1}{5}\log_\sigma n$, in $\mathcal{O}(n/\tau)$ time if T is given in a packed representation.*

As in [50], for a τ-synchronising set A, let $\mathsf{succ}_A(i) := \min\{j \in A \cup \{n-2\tau+2\} : j \ge i\}$.

▶ **Lemma 8** ([50, Fact 3.2]). *If $p = \mathsf{per}(T[i\,..\,i+3\tau-2]) \le \frac{1}{3}\tau$, then $T[i\,..\,\mathsf{succ}_A(i)+2\tau-1)$ is the longest prefix of $T[i\,..\,|T|]$ with period p.*

▶ **Lemma 9** ([50, Fact 3.3]). *If a string U with $|U| \ge 3\tau-1$ and $\mathsf{per}(U) > \frac{1}{3}\tau$ occurs at positions i and j in T, then $\mathsf{succ}_A(i) - i = \mathsf{succ}_A(j) - j \le |U| - 2\tau$.*

A *τ-run* R is a run of length at least $3\tau-1$ with period at most $\frac{1}{3}\tau$. The *Lyndon root* of R is the lexicographically smallest cyclic shift of $R[1\,..\,\mathsf{per}(R)]$. The proof of the following lemma resembles an argument given in [50, Section 6.1.2].

▶ **Lemma 10.** *For a positive integer τ, a string $T \in [0,\sigma)^n$ contains $\mathcal{O}(n/\tau)$ τ-runs. Moreover, if $\tau \le \frac{1}{9}\log_\sigma n$, given a packed representation of T, we can compute all τ-runs in T and group them by their Lyndon roots in $\mathcal{O}(n/\tau)$ time. Within the same complexities, for each τ-run, we can compute the two leftmost occurrences of its Lyndon root.*

Proof. The first claim of the lemma follows by the periodicity lemma (Lemma 5). Recall that a τ-run is a run of length at least $3\tau-1$ with period at most $\frac{1}{3}\tau$. Suppose, towards a contradiction, that two distinct τ-runs overlap by more than $\frac{2}{3}\tau$ positions. By the fact that their periods are at most $\frac{1}{3}\tau$ and an application of the periodicity lemma (Lemma 5), we obtain that these two τ-runs cannot be distinct as they share the same shortest period; a contradiction. This implies that two distinct τ-runs can overlap by no more than $\frac{2}{3}\tau$ positions. In turn, this implies that T contains $\mathcal{O}(n/\tau)$ runs.

Let us now show how to efficiently compute τ-runs. We first compute a τ-synchronising set A in $\mathcal{O}(n/\tau)$ time using Theorem 7. By the definition of such a set, a position $i \in [1, n-3\tau+2]$ in T is a starting position of a τ-run if and only if $[i, i+\tau) \cap A = \emptyset$ and $i-1 \in A$. The period of this τ-run is equal to $p = \mathsf{per}(T[i\,..\,i+3\tau-2])$. Then, by Lemma 8, the longest prefix of $T[i\,..\,n]$ with period p is $T[i\,..\,\mathsf{succ}_A(i)+2\tau-2]$. Using these conditions, we can compute all τ-runs in $\mathcal{O}(n/\tau)$ time in a single scan of A. (We assume that T is concatenated with 3τ occurrences of a letter not occurring in T and ignore any τ-run containing such letters.)

We now show how to group all τ-runs by Lyndon roots. By the conditions of the statement, there are no more than $\sigma^{3\tau} \le n^{1/3}$ distinct strings of length $3\tau-1$. We generate all of them, and for each of them that is periodic we compute its Lyndon root in $\mathcal{O}(n^{1/3}\log_\sigma n)$ time in total [8]. We store these pairs in a table: the index is the string and the value is the Lyndon root. As the Lyndon root of a τ-run $T[i\,..\,j]$ coincides with the Lyndon root of $T[i\,..\,i+3\tau-2]$, we can group all τ-runs as desired using this table in $\mathcal{O}(n/\tau)$ time. This is because $T[i\,..\,i+3\tau-2]$ can be read in $\mathcal{O}(1)$ time in the word RAM model, and so each τ-run is processed in $\mathcal{O}(1)$ time.

Similarly, within the same tabulation process, in $\mathcal{O}(\tau)$ time, for each distinct string S of length $3\tau-1$ with Lyndon root R, we compute and store the position i_R of the leftmost occurrence of R in S [8]. Then, for each τ-run $T[a\,..\,b]$ with period p and Lyndon root R, we can obtain, due to periodicity, in $\mathcal{O}(1)$ time the starting positions $a+i_R-1$ and $a+i_R+p-1$ of the two leftmost occurrences of R in $T[a\,..\,b]$. ◀

▶ **Theorem 11** ([50, Theorem 4.3])**.** *Given a packed representation of T and a τ-synchronising set A of T of size $\mathcal{O}(n/\tau)$ for $\tau = \mathcal{O}(\log_\sigma n)$, we can compute in $\mathcal{O}(n/\tau)$ time the lexicographic order of all suffixes of T starting at positions in A.*

We often want to preprocess a string T to be able to answer queries of the form $\mathsf{LCP}(T[i\mathinner{.\,.} n], T[j\mathinner{.\,.} n])$ [55]. For this case, there exists an optimal data structure that applies synchronising sets.

▶ **Theorem 12** ([50, Theorem 5.4])**.** *Given a packed representation of a string $T \in [0, \sigma)^n$, LCP queries on T can be answered in $\mathcal{O}(1)$ time after $\mathcal{O}(n/\log_\sigma n)$-time preprocessing.*

3 Sublinear-Time LCS

We provide different solutions depending on the length ℓ of an LCS. Lemmas 13, 14, and 19 directly yield Theorem 1.

3.1 Solutions for Short and Long LCS

▶ **Lemma 13.** *The LCS problem can be solved in $\mathcal{O}(n/\log_\sigma n)$ time if $\ell \le \frac{1}{3}\log_\sigma n$.*

Proof. Let us set $m = \lceil \frac{1}{3}\log_\sigma n \rceil$. We use the so-called standard trick: we split both S and T into $\mathcal{O}(n/m)$ fragments, each of length $2m$ (perhaps apart from the last one), starting at positions equal to $1 \bmod m$. For each of the strings, we obtain at most $\sigma^{2m} = \mathcal{O}(n^{2/3})$ *distinct* substrings. For each of the two strings, we can compute the distinct such substrings in $\mathcal{O}(n/m + n^{2/3})$ time; let $S_1, \ldots, S_p$ and $T_1, \ldots, T_q$ be the resulting substrings. Then the problem reduces to computing the LCS of strings $X = S_1\, a_1\, S_2\, a_2 \ldots S_p\, a_p$ and $Y = T_1\, b_1\, T_2\, b_2 \ldots T_q\, b_q$ for distinct letters $a_1, \ldots, a_p, b_1, \ldots, b_q$. Strings X and Y have length $\mathcal{O}(n^{2/3}m)$ and their LCS can be computed in linear time by constructing the suffix tree of $X\#Y$, where $\# \notin \Sigma$ [35]. Thus, the overall time complexity is $\mathcal{O}(n/m + n^{2/3}m) = \mathcal{O}(n/\log_\sigma n)$. ◀

The proof of the following lemma, for the case where an LCS is long, i.e., of length $\ell = \Omega(\frac{\log^4 n}{\log^2 \sigma})$, uses difference covers and the $\mathcal{O}(N \log N)$-time solution to the Two String Families LCP Problem. This proof closely follows [26].

▶ **Lemma 14.** *The LCS problem can be solved in $\mathcal{O}(n/\log_\sigma n)$ time if $\ell = \Omega(\frac{\log^4 n}{\log^2 \sigma})$.*

Before proceeding to the proof, we need to introduce difference covers. We say that a set $D \subseteq \mathbb{Z}_+$ is a *d-cover* if there is a constant-time computable function h such that for positive integers i, j we have $0 \le h(i,j) < d$ and $i + h(i,j), j + h(i,j) \in D$. The following fact synthesises a well-known construction implicitly used in [22], for example.

▶ **Theorem 15** ([56, 22])**.** *For each positive integer d there is a d-cover D such that $D \cap [1, n]$ is of size $\mathcal{O}(\frac{n}{\sqrt{d}})$ and can be constructed in $\mathcal{O}(\frac{n}{\sqrt{d}})$ time.*

Proof of Lemma 14. Let us assume that the answer to LCS is of length $\ell \ge d$ for some parameter d. We first compute a d-cover D of $[1, n]$ in $\mathcal{O}(n/\sqrt{d})$ time. Let us now consider a position i from S and a position j from T. We have that $0 \le h(i,j) < d$ and $i + h(i,j), j + h(i,j) \in D$. Hence, if we want to find an LCS whose length is at least d, we can use the elements of the d-cover as anchors between occurrences of the sought LCS, the length of which equals

$$\max_{i,j \in D} \{\mathsf{LCP}((S[1\mathinner{.\,.} i))^R, (T[1\mathinner{.\,.} j))^R) + \mathsf{LCP}(S[i\mathinner{.\,.} n], T[j\mathinner{.\,.} m])\}. \quad (1)$$

As also done in [26], we set:

- $\mathcal{P} := \{((S[1\,..\,i))^R, S[i\,..\,|S|]) : i \in D\}$, $\mathcal{Q} := \{((T[1\,..\,i))^R, T[i\,..\,|T|]) : i \in D\}$, and
- $\mathcal{F} := \{U : (U,V) \in \mathcal{P} \cup \mathcal{Q}$ or $(V,U) \in \mathcal{P} \cup \mathcal{Q}\}$.

It is then readily verified that, after building the compacted trie $\mathcal{T}(\mathcal{F})$, evaluating Equation (1) reduces to solving the Two String Families LCP Problem for $\mathcal{P}$, $\mathcal{Q}$ and $\mathcal{F}_1 = \mathcal{F}_2 = \mathcal{F}$ with $N = \mathcal{O}(|D|) = \mathcal{O}(n/\sqrt{d})$. In order to build $\mathcal{T}(\mathcal{F})$, due to Lemma 6, it suffices to sort the elements of $\mathcal{F}$ lexicographically and answer LCP queries between consecutive elements in the resulting sorted list. To this end, we first preprocess $S\#_1S^R\#_2T\#_3T^R$, where $\#_i \notin \Sigma$ for all i are distinct letters, in $\mathcal{O}(n/\log_\sigma n)$ time as in Theorem 12, in order to allow for $\mathcal{O}(1)$-time LCP queries. Then, we employ merge sort in order to sort the elements of $\mathcal{F}$, performing each comparison using an LCP query. This step requires $\mathcal{O}(|\mathcal{F}|\log|\mathcal{F}|)$ time. Then, answering LCP queries for consecutive elements requires $\mathcal{O}(|\mathcal{F}|)$ time in total. We finally employ Lemma 3. We have that each of $\mathcal{P}$, $\mathcal{Q}$, and $\mathcal{F}$ is of size at most $N = \mathcal{O}(n/\sqrt{d})$. The overall time complexity is

$$\mathcal{O}\left(N\log N + n/\log_\sigma n\right) = \mathcal{O}(n\log n/\sqrt{d} + n/\log_\sigma n),$$

which is $\mathcal{O}(n/\log_\sigma n)$ if $d = \Omega(\frac{\log^4 n}{\log^2 \sigma})$. ◀

▶ Remark 16. The only modification compared to the solution of [26] lies in the construction of $\mathcal{T}(\mathcal{F})$. In [26], $\mathcal{T}(\mathcal{F})$ is extracted from the generalised suffix tree of strings S, T, S^R, and T^R, which we cannot afford to construct, as its construction requires $\Omega(n)$ time.

3.2 Solution for Medium-Length LCS

We now give a solution to the LCS problem for $\ell \in [\frac{1}{3}\log_\sigma n, 2^{\sqrt{\log n}}]$. We first construct three subsets of positions in $S\$T$, where $\$ \notin \Sigma$, of size $\mathcal{O}(n/\log_\sigma n)$ as follows. For $\tau = \lfloor \frac{1}{9}\log_\sigma n\rfloor$, let A_I be a τ-synchronising set of $S\$T$. For each τ-run in $S\$T$, we insert to A_{II} the starting positions of the first two occurrences of the Lyndon root of the τ-run and to A_{III} the last position of the τ-run. The elements of A_{II} and A_{III} store the τ-run they originate from. Finally, we denote $A^S_j = A_j \cap [1,|S|]$ and $A^T_j = \{a - |S| - 1 : a \in A_j,\, a > |S| + 1\}$ for $j = I, II, III$. The following lemma shows that there exists an LCS of S and T for which $A_I \cup A_{II} \cup A_{III}$ is a set of *anchors* that satisfies certain distance requirements.

▶ **Lemma 17.** *If an LCS of S and T has length $\ell \geq 3\tau$, then there exist positions $i^S \in [1,|S|]$, $i^T \in [1,|T|]$, a shift $\delta \in [0,\ell)$, and $j \in \{I, II, III\}$ such that $S[i^S\,..\,i^S+\ell) = T[i^T\,..\,i^T+\ell)$, $i^S+\delta \in A^S_j$, $i^T+\delta \in A^T_j$, and*

- *if $j = I$, then $\delta \in [0,\tau)$;*
- *if $j = II$, then $S[i^S\,..\,i^S+\ell)$ is contained in the τ-run from which $i^S+\delta \in A^S$ originates;*
- *if $j = III$, then $\delta \geq 3\tau-1$ and $S[i^S\,..\,i^S+\delta]$ is a suffix of the τ-run from which $i^S+\delta \in A^S$ originates.*

Proof. By the assumption, there exist $i^S \in [1,|S|]$ and $i^T \in [1,|T|]$ such that $S[i^S\,..\,i^S+\ell) = T[i^T\,..\,i^T+\ell)$. Let us choose any such pair (i^S, i^T) minimising the sum $i^S + i^T$. We have the following cases.

1. If $\mathsf{per}(S[i^S\,..\,i^S+3\tau-2]) > \frac{1}{3}\tau$, then, by the definition of a τ-synchronising set, in this case there exist some elements $a^S \in A^S_I \cap [i^S, i^S+\tau)$ and $a^T \in A^T_I \cap [i^T, i^T+\tau)$. Let us choose the smallest such elements. By Lemma 9, we have $a^S - i^S = a^T - i^T$.
2. Else, $p = \mathsf{per}(S[i^S\,..\,i^S+3\tau-2]) \leq \frac{1}{3}\tau$. We have two subcases.

a. If $p = \mathsf{per}(S[i^S \mathinner{.\,.} i^S + \ell))$, then, by the choice of i^S and i^T there exists a τ-run R_S in S that starts at position in $(i^S - p \mathinner{.\,.} i^S]$ and a τ-run R_T in T that starts at position in $(i^T - p \mathinner{.\,.} i^T]$. Moreover, by Lemma 5, both runs have equal Lyndon roots. For each $X \in \{S, T\}$, let us choose a^X as the leftmost starting position of a Lyndon root of R_X that is $\geq i^X$. We have $a^S - i^S = a^T - i^T \in [0, \frac{1}{3}\tau)$. Each position a^X will be the starting position of the first or the second occurrence of the Lyndon root of R_S, so $a^S \in A^S_{II}$ and $a^T \in A^T_{II}$.

b. Else, $p \neq \mathsf{per}(S[i^S \mathinner{.\,.} i^S + \ell))$. We have $d = \min\{b \geq p : S[i^S + b] \neq S[i^S + b - p]\} < \ell$ (and $d \geq 3\tau - 1$). In this case, $a^S = i^S + d - 1$ and $a^T = i^T + d - 1$ are the ending positions of τ-runs with period p in S and T, respectively, so $a^S \in A^S_{III}$ and $a^T \in A^T_{III}$. ◀

Case $j = I$ from the above lemma corresponds to the Two String Families LCP Problem with $\mathcal{P}$ and $\mathcal{Q}$ being $(\tau, 2^{\sqrt{\log n}})$-families. Let us introduce a variant of the Two String Families LCP Problem that intuitively corresponds to the case $j \in \{II, III\}$. A family of string pairs $\mathcal{P}$ is called a *prefix family* if there exists a string Y such that, for each $(U, V) \in \mathcal{P}$, U is a prefix of Y. We arrive at this special case with first components of $\mathcal{P}$ and $\mathcal{Q}$ being prefixes of some cyclic shift of a power of a (common) Lyndon root of τ-runs.

▶ **Lemma 18.** *An instance of the Two String Families LCP Problem in which $\mathcal{P} \cup \mathcal{Q}$ is a prefix family can be solved in $\mathcal{O}(N)$ time.*

Proof. By traversing $\mathcal{T}(\mathcal{F}_2)$ we can compute in $\mathcal{O}(N)$ time a list $\mathcal{R}$ being a union of sets $\mathcal{P}$ and $\mathcal{Q}$ in which the second components are ordered lexicographically.

Consider an element $e = (U, V) \in \mathcal{P}$. Let $\mathsf{lex\text{-}pred}(e) = (Y_1, Y_2)$ be the predecessor of e in $\mathcal{R}$ that originates from $\mathcal{Q}$ and satisfies $|Y_1| \geq |U|$. Similarly, let $\mathsf{lex\text{-}succ}(e) = (Z_1, Z_2)$ be the successor of e in $\mathcal{R}$ that originates from $\mathcal{Q}$ and satisfies $|Z_1| \geq |U|$ (inspect Figure 1). Further, let $\lambda(e) = \max\{\mathsf{LCP}(U, Y_1) + \mathsf{LCP}(U, Y_2), \mathsf{LCP}(V, Z_1) + \mathsf{LCP}(V, Z_2)\}$. We also define the same notations for $e \in \mathcal{Q}$ with $\mathcal{P}$ and $\mathcal{Q}$ swapped around.

▷ Claim. $\mathrm{maxPairLCP}(\mathcal{P}, \mathcal{Q}) = \max_{e \in \mathcal{P} \cup \mathcal{Q}} \lambda(e)$.

Proof. First, we clearly have $\mathrm{maxPairLCP}(\mathcal{P}, \mathcal{Q}) \geq \max_{e \in \mathcal{P} \cup \mathcal{Q}} \lambda(e)$.

Let $(P_1, P_2) \in \mathcal{P}$ and $(Q_1, Q_2) \in \mathcal{Q}$ be such that

$$\mathsf{LCP}(P_1, Q_1) + \mathsf{LCP}(P_2, Q_2) = \mathrm{maxPairLCP}(\mathcal{P}, \mathcal{Q}).$$

Without loss of generality, let us assume that $|P_1| \leq |Q_1|$ and that (Q_1, Q_2) precedes (P_1, P_2) in $\mathcal{R}$.

Let $(Y_1, Y_2) = \mathsf{lex\text{-}pred}((P_1, P_2))$. Now, we have $\mathsf{LCP}(P_1, Y_1) = |P_1| \geq \mathsf{LCP}(P_1, Q_1)$, since P_1 and Y_1 are prefixes of the same string and $|P_1| \leq |Y_1|$. Further, we have $\mathsf{LCP}(P_2, Y_2) \geq \mathsf{LCP}(P_2, Q_2)$ since (Q_1, Q_2), (Y_1, Y_2), and (P_1, P_2) appear in $\mathcal{R}$ in this order. Thus, we get

$$\mathrm{maxPairLCP}(\mathcal{P}, \mathcal{Q}) = \mathsf{LCP}(P_1, Q_1) + \mathsf{LCP}(P_2, Q_2) \leq \mathsf{LCP}(P_1, Y_1) + \mathsf{LCP}(P_2, Y_2) \leq \max_{e \in \mathcal{P} \cup \mathcal{Q}} \lambda(e),$$

which concludes the proof of this claim. ◁

To compute $\mathsf{lex\text{-}succ}(e)$ and $\mathsf{lex\text{-}pred}(e)$ for each $e \in \mathcal{P} \cup \mathcal{Q}$ we proceed as follows. We process the list $\mathcal{R}$ in the order of non-decreasing lengths with respect to the first component. After processing all elements e of some length, we remove them from $\mathcal{R}$.

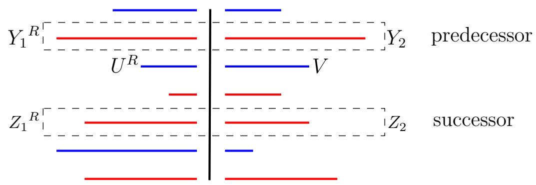


Figure 1 The setting in Lemma 18 on list $\mathcal{R}$. With red colour we denote the elements of $\mathcal{P}$ and with blue colour the elements of $\mathcal{Q}$. For element $e = (U, V)$ from $\mathcal{Q}$, we have that Y_1 from $\mathcal{P}$ is the predecessor with $|Y_1| \geq |U|$ and Z_1 from $\mathcal{P}$ is the successor with $|Z_1| \geq |U|$.

We maintain the list $\mathcal{R}$ using the data structure of Gabow and Tarjan [40] for a special case of the union-find problem, in which the sets are formed by consecutive integers. Let us think of elements originating from $\mathcal{P}$ to be coloured by red and elements originating from $\mathcal{Q}$ to be coloured by blue. The sets the union-find data structure is initialised with correspond to maximal sequences of elements of the same colour in $\mathcal{R}$. Each set has as an id its smallest element. Each set also maintains the following satellite information: pointers to the head and to the tail of a list of all non-deleted elements in this set. Every element of $\mathcal{R}$ stores a pointer to the element in the list in which it is represented. It can thus be deleted at any moment in $\mathcal{O}(1)$ time. When the satellite list of a set S_i becomes empty, we union S_{i-1}, S_i, and S_{i+1}. To find $\mathsf{pred}(e)$ we perform the following; the procedure of $\mathsf{succ}(e)$ is analogous. We find the set S_i to which e belongs using a find operation. Let the id of S_i be α. By using another find operation to find $\alpha - 1$, we find the set S_{i-1}. The tail of the list of S_{i-1} is $\mathsf{pred}(e)$; in the analogous procedure, the head of the list of S_{i+1} is $\mathsf{succ}(e)$. The algorithm is correct because we process $\mathcal{R}$ in the order of non-decreasing lengths. Each union or find operation requires $\mathcal{O}(1)$ amortized time [40]. Thus, this procedure takes $\mathcal{O}(N)$ time in total.

Finally, we preprocess the compacted tries $\mathcal{T}(\mathcal{F}_1)$ and $\mathcal{T}(\mathcal{F}_2)$ in $\mathcal{O}(N)$ time to be able to answer lowest common ancestor (LCA) queries in $\mathcal{O}(1)$ time [14]. For any $e = (U, V)$, given $\mathsf{lex\text{-}pred}(e)$ and $\mathsf{lex\text{-}succ}(e)$ we can compute $\lambda(e)$ in $\mathcal{O}(1)$ time by answering LCP queries using the LCA data structure. ◀

We are now ready to state the main result of this subsection.

▶ **Lemma 19.** *The LCS problem can be solved in $\mathcal{O}(n \log \sigma / \sqrt{\log n})$ time using $\mathcal{O}(n / \log_\sigma n)$ space, provided that $\ell \in [\frac{1}{3} \log_\sigma n, 2^{\sqrt{\log n}}]$.*

Proof. Recall that $\tau = \lfloor \frac{1}{9} \log_\sigma n \rfloor$. The set of anchors $A = A_I \cup A_{II} \cup A_{III}$ consists of a τ-synchronising set and of $\mathcal{O}(1)$ positions per each τ-run in $S\$T$. Hence, $|A| = \mathcal{O}(n/\tau)$ and A can be constructed in $\mathcal{O}(n/\tau)$ time by Theorem 7 and Lemma 10.

We construct sets of pairs of substrings of $X = S\$_1T\$_2S^R\$_3T^R$. First, for $\Delta = \lfloor 2^{\sqrt{\log n}} \rfloor$:

$$\mathcal{P}_I = \{((S[a - \tau \mathinner{.\,.} a))^R, S[a \mathinner{.\,.} a + \Delta)) \,:\, a \in A_I^S\}.$$

Then, for each group $\mathcal{G}$ of τ-runs in S and T with equal Lyndon root, we construct the following set of string pairs:

$$\mathcal{P}_{II}^{\mathcal{G}} = \{((S[x \mathinner{.\,.} a))^R, S[a \mathinner{.\,.} y)) \,:\, a \in A_{II}^S \text{ that originates from } \tau\text{-run } S[x \mathinner{.\,.} y] \in \mathcal{G}\}.$$

We define the *tail* of a τ-run $S[i \mathinner{.\,.} j]$ with period p and Lyndon root $S[i' \mathinner{.\,.} i' + p)$ as $(j + 1 - i') \bmod p$ (and same for τ-runs in T). For each group of τ-runs in S and T with equal Lyndon roots, we group the τ-runs belonging to it by their tails. This can be done in

$\mathcal{O}(n/\tau)$ time using tabulation, since the tail values are up to $\frac{1}{3}\tau$. For each group $\mathcal{G}$ of τ-runs in S and T with equal Lyndon root and tail, we construct the following set of string pairs:

$$\mathcal{P}_{III}^{\mathcal{G}} = \{((S[x\mathinner{.\,.} y))^R, S[y\mathinner{.\,.}|S|]) \,:\, S[x\mathinner{.\,.} y] \in \mathcal{G}\}.$$

Simultaneously, we create sets $\mathcal{Q}_I$, $\mathcal{Q}_{II}^{\mathcal{G}}$ and $\mathcal{Q}_{III}^{\mathcal{G}}$ defined with T instead of S.

Now, it suffices the output the maximum of maxPairLCP$(\mathcal{P}_I, \mathcal{Q}_I)$, maxPairLCP$(\mathcal{P}_{II}^{\mathcal{G}}, \mathcal{Q}_{II}^{\mathcal{G}})$, and maxPairLCP$(\mathcal{P}_{III}^{\mathcal{G}}, \mathcal{Q}_{III}^{\mathcal{G}})$, where $\mathcal{G}$ ranges over groups of τ-runs. Computing any individual maxPairLCP value can be expressed as an instance of the Two String Families LCP Problem provided that all the first and second components of families are represented as nodes of compacted tries. We will use Lemma 6 to construct these compacted tries. LCP queries can be answered efficiently due to Theorem 12, so it suffices to be able to sort all the first and second components of each pair of string pair sets lexicographically. Each of the sets $\mathcal{P}_I$, $\mathcal{Q}_I$ can be ordered by the second components using Theorem 11 since A_I is a τ-synchronising set, and by the first components with easy preprocessing using the fact that the number of possible τ-length strings is $\sigma^\tau = \mathcal{O}(n^{1/9})$. In a set $\mathcal{P}_{II}^{\mathcal{G}}$, both all first and all second components are prefixes of a single string (a power of the common Lyndon root). Hence, they can be sorted simply by comparing their lengths. This sorting is performed simultaneously for all the families $\mathcal{P}_{II}^{\mathcal{G}}$, $\mathcal{Q}_{II}^{\mathcal{G}}$ in $\mathcal{O}(n/\tau)$ time via radix sort. Finally, to sort the second components of the sets $\mathcal{P}_{III}^{\mathcal{G}}$ $\mathcal{Q}_{III}^{\mathcal{G}}$, instead of comparing strings of the form $S[y\mathinner{.\,.}|S|]$ (and same for T), we can equivalently compare strings $S[y-2\tau+1\mathinner{.\,.}|S|]$ which are known to start at positions from a τ-synchronising set by Lemma 8. This sorting is done across all groups using radix sort and Theorem 11. The correctness follows by Lemma 17.

Finally, we observe that $(\mathcal{P}_I, \mathcal{Q}_I)$ is a (τ, Δ)-family of size $N = \mathcal{O}(n/\tau)$, and thus maxPairLCP$(\mathcal{P}_I, \mathcal{Q}_I)$ can be computed in $\mathcal{O}(n \log \sigma / \sqrt{\log n})$ time and $\mathcal{O}(n/\log_\sigma n)$ space using Lemma 4. On the other hand, $(\mathcal{P}_{II}^{\mathcal{G}}, \mathcal{Q}_{II}^{\mathcal{G}})$ and $(\mathcal{P}_{III}^{\mathcal{G}}, \mathcal{Q}_{III}^{\mathcal{G}})$ are prefix families of total size $\mathcal{O}(n/\tau)$, so the corresponding instances of the Two String Families LCP Problem can be solved in $\mathcal{O}(n/\log_\sigma n)$ total time using Lemma 18. ◀

4 Proof of Lemma 4 via Wavelet Trees

Wavelet trees. For an arbitrary alphabet Σ, the *skeleton tree* for Σ is a full binary tree $\mathcal{T}$ together with a bijection between Σ and the leaves of $\mathcal{T}$. For a node $v \in \mathcal{T}$, let Σ_v denote the subset of Σ that corresponds to the leaves in the subtree of v.

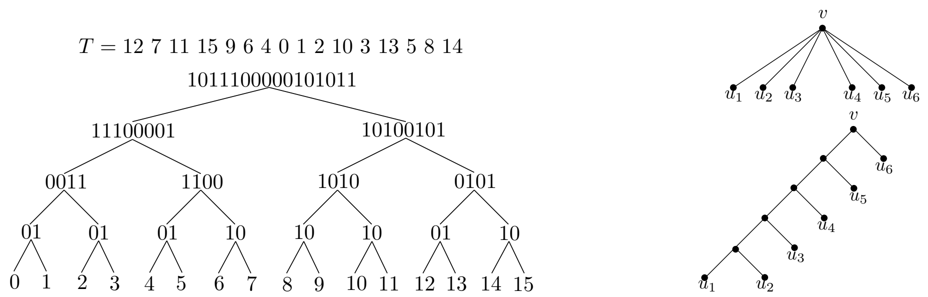
The *$\mathcal{T}$-shaped wavelet tree* of a string $T \in \Sigma^*$ consists of bit vectors assigned to internal nodes of $\mathcal{T}$ (inspect Figure 2(a)). For an internal node v with children v_L and v_R, let T_v denote the maximal subsequence of T that consists of letters from Σ_v; the bit vector $B_v[1\mathinner{.\,.}|T_v|]$ is defined so that $B_v[i] = 0$ if $T_v[i] \in \Sigma_{v_L}$ and $B_v[i] = 1$ if $T_v[i] \in \Sigma_{v_R}$.

Wavelet trees were introduced in [44], whereas efficient construction algorithms were presented in [58, 5].

▶ **Theorem 20** (see [58, Theorem 2])**.** *Given the packed representation of a string $T \in [0, \sigma)^n$ and a skeleton tree $\mathcal{T}$ of height h, the $\mathcal{T}$-shaped wavelet tree of T takes $\mathcal{O}(nh/\log n + \sigma)$ space and can be constructed in $\mathcal{O}(nh/\sqrt{\log n} + \sigma)$ time.*

Wavelet trees are sometimes constructed for sequences $T \in \mathcal{M}^*$ over an alphabet $\mathcal{M} \subseteq \Sigma^*$ that itself consists of strings (see e.g. [50]). In this case, the skeleton tree $\mathcal{T}$ is often chosen to resemble the compacted trie of $\mathcal{M}$. Formally, we say that a skeleton tree $\mathcal{T}$ for $\mathcal{M}$ is *prefix-consistent* if each node $v \in \mathcal{T}$ admits a label $\mathsf{val}(v) \in \Sigma^*$ such that:

- if v is a leaf, then $\mathsf{val}(v)$ is the corresponding string in $\mathcal{M}$;



(a) Standard skeleton tree with $|T| = \sigma = 16$.

(b) Binarisation with height $\mathcal{O}(\alpha + \log N)$.

Figure 2 (a) Let v be left child of the root node. Then $\Sigma_v = \{0, 1, \dots, 7\}$, $T_v = 7, 6, 4, 0, 1, 2, 3, 5$ and so $B_v = 11100001$: $7, 6, 4, 5$ belong to the right subtree of v and $0, 1, 2, 3$ to the left. (b) For each i, let the size of the subtree rooted at u_i be 2^i. The binarisation from [5] leads to height $\mathcal{O}(\alpha + \log N)$, favouring heavier children.

- if v is a node with children v_L, v_R, then, for all leaves u_L, u_R in the subtrees of v_L and v_R, respectively, the string $\mathsf{val}(v)$ is the longest common prefix of $\mathsf{val}(u_L)$ and $\mathsf{val}(u_R)$.

Observe that if $\mathcal{M} \subseteq \{0,1\}^\alpha$ for some integer α, then the compacted trie $\mathcal{T}(\mathcal{M})$ is a prefix-consistent skeleton tree for $\mathcal{M}$. For larger alphabets, we binarise $\mathcal{T}(\mathcal{M})$ as follows:

▶ **Lemma 21.** *Given the compacted trie $\mathcal{T}(\mathcal{M})$ of a set $\mathcal{M} \subseteq \Sigma^\alpha$, a prefix-consistent skeleton tree of height $\mathcal{O}(\alpha + \log |\mathcal{M}|)$ can be constructed in $\mathcal{O}(|\mathcal{M}|)$ time, with each node v associated to a node v' of $\mathcal{T}(\mathcal{M})$ such that $\mathsf{val}(v) = \mathsf{val}(v')$.*

Proof. We use [5, Corollary 3.2], where the authors showed that any rooted tree of size m and height h can be binarised in $\mathcal{O}(m)$ time so that the resulting tree is of height $\mathcal{O}(h + \log m)$. For $\mathcal{T}(\mathcal{M})$, we obtain height $\mathcal{O}(\alpha + \log |\mathcal{M}|)$ and time $\mathcal{O}(|\mathcal{M}|)$ (inspect Figure 2(b)). ◀

▶ **Lemma 4.** *An instance of the* Two String Families LCP Problem *in which $\mathcal{P}$ and $\mathcal{Q}$ are (α, β)-families can be solved in time $\mathcal{O}(N(\alpha + \log N)(\log \beta + \sqrt{\log N}) / \log N)$ and space $\mathcal{O}(N + N\alpha / \log N)$.*

Proof. By traversing $\mathcal{T}(\mathcal{F}_2)$ we can compute in $\mathcal{O}(N)$ time a list $\mathcal{R}$ being a union of sets $\mathcal{P}$ and $\mathcal{Q}$ in which the second components are ordered lexicographically. We also store a bit vector G of length $|\mathcal{R}|$ that determines, for each element of $\mathcal{R}$, which of the sets $\mathcal{P}$, $\mathcal{Q}$ it originates from. We construct the wavelet tree of the sequence of strings being the first components of pairs from $\mathcal{R}$ using Theorem 20 and the skeleton tree from Lemma 21. Before the wavelet tree is constructed, we pad each string with a symbol $\$ \notin \Sigma$ to make them all of length α; we will ignore the nodes of the wavelet tree with a path-label containing a \$.

For a sublist $\mathcal{X} = (U_1, V_1), \dots, (U_m, V_m)$ of $\mathcal{R}$, by $\mathsf{LCPs}(\mathcal{X})$ we denote the representation of the list $0, \mathsf{LCP}(V_1, V_2), \dots, \mathsf{LCP}(V_{m-1}, V_m)$ as a packed string over alphabet $[0, \beta]$ in space $\mathcal{O}(N / \log_\beta N)$. Together with each $\mathsf{LCPs}(\mathcal{X})$ we also store the bit vector $G(\mathcal{X})$ of origins of elements of $\mathcal{X}$ without increasing the complexity. The list $\mathsf{LCPs}(\mathcal{R})$ can be computed in $\mathcal{O}(N)$ time when constructing $\mathcal{R}$. For each node v of the wavelet tree, we wish to compute $\mathcal{L}_v = \mathsf{LCPs}(\mathcal{R}_v)$, where $\mathcal{R}_v$ is the sublist of $\mathcal{R}$ composed of elements whose first component is in the leaf list Σ_v of v. We will construct the lists $\mathsf{LCPs}(\mathcal{R}_v)$ without actually computing $\mathcal{R}_v$.

Computation of LCPs lists. The lists are computed recursively using the bit vectors from the wavelet tree. Assume we have computed $\mathcal{L}_u$ and wish to compute $\mathcal{L}_v$ for the left child v of u—the computations for the right child are symmetric.

Let $c \in (0,1)$ be a constant. Let us partition $\mathcal{L}_u$ into blocks of $\lambda = \max(1, \lfloor c \log_\beta N \rfloor)$ LCP values. We will process the blocks in order, constructing $\mathcal{L}_v$. Each block of $\mathcal{L}_u$ can be represented in a single word and this representation can be extracted from the packed representation of $\mathcal{L}_u$ in $\mathcal{O}(1)$ time. For each block $W = \mathcal{L}_u(a\lambda \mathinner{.\,.} (a+1)\lambda]$, we retrieve in $\mathcal{O}(1)$ time the corresponding block $D = B_u(a\lambda \mathinner{.\,.} (a+1)\lambda]$ in the bit vector from the wavelet tree. Further, we store $\mu_a = \min \mathcal{L}_v(p_a \mathinner{.\,.} a\lambda]$, where $p_a = \max\{i \in [0, a\lambda) : i = 0 \text{ or } B_u[i] = 0\}$. Let us show how, given W, D and μ_a, we can determine μ_{a+1} and the chunk of $\mathcal{L}_v$ that corresponds to $i \in [1, \lambda]$ such that $D[i] = 0$. The calculations are based on the following well-known fact.

▶ **Fact 22.** *If U_1, U_2, U_3 are strings such that $U_1 \le U_2 \le U_3$, then we have* $\mathsf{LCP}(U_1, U_3) = \min(\mathsf{LCP}(U_1, U_2), \mathsf{LCP}(U_2, U_3))$.

For each $i \in [1, \lambda]$ such that $D[i] = 0$, in increasing order, if a previous position j with $D[j] = 0$ exists, then $\min W(i' \mathinner{.\,.} i]$ should be appended to $\mathcal{L}_v$, where i' is the predecessor of i satisfying $D[i'] = 0$, and otherwise $\min(\{\mu_a\} \cup W[1 \mathinner{.\,.} i])$ should. Then, for the last position $i \in [1, \lambda]$ such that $D[i] = 0$, $\mu_{a+1} = \min W(i \mathinner{.\,.} \lambda]$, and if no such position exists, then $\mu_{a+1} = \min(\{\mu_a\} \cup W[1 \mathinner{.\,.} \lambda])$.

If $\lambda = 1$, the calculations can be performed in $\mathcal{O}(1)$ time. Otherwise, we make use of preprocessing: for every possible combination of (W, D, μ_a), i.e., up to $2c \log N + \log \beta < 3c \log N$ bits, precompute the block to be appended to $\mathcal{L}_v$ and μ_{a+1}, i.e., up to $c \log N + \log \beta < 2c \log N$ bits. We can choose $c > 0$ small enough to make the preprocessing $\mathcal{O}(N^{1-\varepsilon})$ for some $\varepsilon > 0$. Thus, the computation takes $\mathcal{O}(|\mathcal{L}_u|/\lambda)$ time. Within this time, we can also populate the bit vector of origins for v.

Application of LCPs lists. For each node u of the wavelet tree, we must extract the maximum LCP between suffixes of different origins, add the string-depth $|\mathsf{val}(u)|$, and compare the result with the stored candidate. The former can be computed in $\mathcal{O}(|\mathcal{L}_u|/\log_\beta N)$ time as follows. Due to Fact 22, the answer will be the LCP between a pair of second components of consecutive elements of $\mathcal{R}_u$ that originate from different sets, i.e. $\max\{\mathcal{L}_u[i] : i \in [2, |\mathcal{R}_u|], G(\mathcal{R}_u)[i-1] \neq G(\mathcal{R}_u)[i]\}$. We can cover all pairs of consecutive elements of $\mathcal{L}_u$ using $\Theta(1 + |\mathcal{L}_u|/\log_\beta N)$ blocks of $\max(2, \lfloor c \log_\beta N \rfloor)$ LCP values. Each such block, augmented with its corresponding bit vector of origins, consists of at most $2c \log N$ bits. We can thus precompute all possible answers in $\mathcal{O}(N^{1-\epsilon})$ time, and then process each node u in $\Theta(1 + |\mathcal{L}_u|/\log_\beta N)$ time.

Time complexity. The wavelet tree can be built in $\mathcal{O}(Nh/\sqrt{\log N})$ time using space $\mathcal{O}(Nh/\log N)$ by Theorem 20, where $h = \mathcal{O}(\alpha + \log N)$. Computing LCPs for children of a single node u takes $\mathcal{O}(1 + |\mathcal{L}_u|/\log_\beta N)$ time; over all nodes, this is $\mathcal{O}(Nh \log \beta / \log N)$. ◀

5 Faster k-LCS

In this section, we present our $\mathcal{O}(n \log^{k-1/2} n)$-time algorithm for the k-LCS problem with $k = \mathcal{O}(1)$, that underlies Theorem 2. For simplicity, we focus on computing the length of a k-LCS; an actual pair of strings forming a k-LCS can be recovered easily from our approach. If the length of an LCS of S and T is d, then the length of a k-LCS of S and T is in $[d, (k+1)d + k]$. Below, we show how to compute a k-LCS provided that it belongs to an interval $(\ell/2, \ell]$ for a specified ℓ; it is sufficient to call this subroutine $\mathcal{O}(\log k) = \mathcal{O}(1)$ times.

Similarly to our solutions for long and medium-length LCS, we first distinguish anchors $A^S \subseteq [1, |S|]$ in S and $A^T \subseteq [1, |T|]$ in T, as summarised in the following lemma.

▶ **Lemma 23.** *Consider an instance of the k-LCS problem for $k = \mathcal{O}(1)$ and let $\ell \in [1, n]$. In $\mathcal{O}(n)$ time, one can construct sets $A^S \subseteq [1, |S|]$ and $A^T \subseteq [1, T]$ of size $\mathcal{O}(\frac{n}{\ell})$ satisfying the following condition: If a k-LCS of S and T has length $\ell' \in (\ell/2, \ell]$, then there exist positions $i^S \in [1, |S|]$, $i^T \in [1, |T|]$ and a shift $\delta \in [0, \ell')$ such that $i^S + \delta \in A^S$, $i^T + \delta \in A^T$, and the Hamming distance between $S[i^S \mathinner{.\,.} i^S + \ell')$ and $T[i^T \mathinner{.\,.} i^T + \ell')$ is at most k.*

Proof. As in [29], we say that position a in a string X is a *misperiod* with respect to a substring $X[i \mathinner{.\,.} j)$ if $X[a] \neq X[b]$, where b is the unique position such that $b \in [i, j)$ and $(j - i) \mid (b - a)$; for example, $j - i$ is a period of X if and only if there are no misperiods with respect to $X[i \mathinner{.\,.} j)$. We define the set $\mathsf{LeftMisper}_k(X, i, j)$ as the set of k maximal misperiods that are smaller than i and $\mathsf{RightMisper}_k(X, i, j)$ as the set of k minimal misperiods that are not smaller than j. Either set may have fewer than k elements if the corresponding misperiods do not exist. Further, let us define $\mathsf{Misper}_k(X, i, j) = \mathsf{LeftMisper}_k(X, i, j) \cup \mathsf{RightMisper}_k(X, i, j)$ and $\mathsf{Misper}(X, i, j) = \bigcup_{k=0}^{\infty} \mathsf{Misper}_k(X, i, j)$.

Similar to Lemma 17, we construct three subsets of positions in $Y = \#S\$T$, where $\#, \$ \notin \Sigma$. For $\tau = \lfloor \ell/(6(k + 1)) \rfloor$, let A_I be a τ-synchronising set of Y. Let $Y[i \mathinner{.\,.} j]$ be a τ-run with period p and assume that the first occurrence of its Lyndon root is at a position q of Y. Then, for $Y[i \mathinner{.\,.} j]$, for each $x \in \mathsf{LeftMisper}_{k+1}(Y, i, i + p)$, we insert to A_{II} the two smallest positions in $[x + 1, |Y|]$ that are equivalent to $q \pmod p$. Moreover, we insert to A_{III} the positions in $\mathsf{Misper}_{k+1}(Y, i, i + p)$. Finally, we denote $A = A_I \cup A_{II} \cup A_{III}$, as well as $A^S = \{a - 1 : a \in A \cap [2, |S| + 1]\}$ and $A^T = \{a - |S| - 2 : a \in A \cap [|S| + 3, |Y|]\}$. The proof of the following claim resembles that of Lemma 17.

▷ Claim 24. The sets A^S and A^T satisfy the condition stated in Lemma 23.

Proof. For two strings $U, V \in \Sigma^n$, we define the *mismatch positions* $\mathsf{MP}(U, V) = \{i \in [1, n] : U[i] \neq V[i]\}$. Moreover, for $k \in \mathbb{Z}_{\geq 0}$, we write $U =_k V$ if $|\mathsf{MP}(U, V)| \leq k$.

By the assumption of Lemma 23, a k-LCS of S and T has length $\ell' \in (\ell/2, \ell]$. Consequently, there exist $i^S \in [1, |S|]$ and $i^T \in [1, |T|]$ such that $U =_k V$, for $U = S[i^S \mathinner{.\,.} i^S + \ell')$ and $V = T[i^T \mathinner{.\,.} i^T + \ell')$. Let us choose any such pair (i^S, i^T) minimising the sum $i^S + i^T$. We shall prove that there exists $\delta \in [0, \ell')$ such that $i^S + \delta \in A^S$ and $i^T + \delta \in A^T$. In each of the three main cases, we actually show that $i^S + \delta \in A^S_j$ and $i^T + \delta \in A^T_j$ for $\delta \in [0, \ell')$ and for some $j \in \{I, II, III\}$, where $A^S_j = \{a - 1 : a \in A_j \cap [2, |S| + 1]\}$ and $A^T_j = \{a - |S| - 2 : a \in A_j \cap [|S| + 3, |Y|]\}$ (recall that $Y = \#S\$T$).

Recall that $\tau = \lfloor \ell/(6(k + 1)) \rfloor$. There exists a shift $s \in [0, \ell' - 3\tau + 2]$ such that

$$W := S[i^S + s \mathinner{.\,.} i^S + s + 3\tau - 2] = T[i^T + s \mathinner{.\,.} i^T + s + 3\tau - 2].$$

First, assume that $\mathsf{per}(W) > \frac{1}{3}\tau$. By the definition of a τ-synchronising set, in this case there exist some elements $a^S \in A^S_I \cap [i^S + s, i^S + s + \tau)$ and $a^T \in A^T_I \cap [i^T + s, i^T + s + \tau)$. Let us choose the smallest such elements. By Lemma 9, we have $a^S - i^S = a^T - i^T$.

From now on we consider the case that $p = \mathsf{per}(W) \leq \frac{1}{3}\tau$. Thus, each of the two distinguished fragments of S and T that match W belongs to a τ-run with period p.

In the case where $\mathsf{Misper}_{k+1}(U, 1 + s, 1 + s + p) \cap \mathsf{Misper}_{k+1}(V, 1 + s, 1 + s + p) \neq \emptyset$, there exist $a^S \in A^S_{III}$ and $a^T \in A^T_{III}$ that satisfy the desired condition. This is because, for any string Z and its run $Z[i \mathinner{.\,.} j]$ with period p and $i' \in [i, j - p + 1]$, $\mathsf{Misper}_{k+1}(Z, i, i + p) = \mathsf{Misper}_{k+1}(Z, i', i' + p)$.

In order to handle the complementary case, we rely on the following claim. We recall its short proof for completeness.

▷ Claim 25 ([29, Lemma 13]). Assume that $U =_k V$ and that $U[i\,..\,j) = V[i\,..\,j)$. Let

$$I = \mathsf{Misper}_{k+1}(U, i, j) \text{ and } I' = \mathsf{Misper}_{k+1}(V, i, j).$$

If $I \cap I' = \emptyset$, then $\mathsf{MP}(U, V) = I \cup I'$, $I = \mathsf{Misper}(U, i, j)$, and $I' = \mathsf{Misper}(V, i, j)$.

Proof. Let $J = \mathsf{Misper}(U, i, j)$ and $J' = \mathsf{Misper}(V, i, j)$. We first observe that $I \cup I' \subseteq \mathsf{MP}(U, V)$ since $I \cap I' = \emptyset$. Then, $U =_k V$ implies that $|\mathsf{MP}(U, V)| \le k$ and hence $|I| \le k$ and $|I'| \le k$, which in turn implies that $I = J$ and $I' = J'$. The observation that $\mathsf{MP}(U, V) \subseteq J \cup J'$ concludes the proof. ◁

Towards a contradiction, let us suppose that there are no $a^S \in A^S_{II}$ and $a^T \in A^T_{II}$ satisfying the desired condition. By Claim 25, we have

$$|\mathsf{LeftMisper}_{k+1}(U, 1+s, 1+s+p)|, |\mathsf{LeftMisper}_{k+1}(V, 1+s, 1+s+p)| \le k.$$

Therefore, at least one of the following holds:

$$i^S \in \mathsf{Misper}_{k+1}(\#S, 1+i^S+s, 1+i^S+s+p) \text{ or } i^T \in \mathsf{Misper}_{k+1}(\$T, 1+i^T+s, 1+i^T+s+p);$$

otherwise, $S[i^S - 1] = T[i^T - 1]$ provides a contradiction. Let us assume that the first of these two conditions holds; the other case is symmetric. Then, $[i^S, i^S + 2p)$ contains two elements of A^S_{II}. If $A^T_{II} \cap [i^T, i^T + 2p) = \emptyset$, then this implies that

$$[i^T - p + 1, i^T] \cap \mathsf{Misper}_{k+1}(\$T, 1 + i^T + s, 1 + i^T + s + p) = \emptyset.$$

In particular, we have $i^T > p$. Now, let us consider U and $V' = T[i^T - p\,..\,i^T + \ell' - p)$. By Claim 25, we have that $|\mathsf{Misper}(U, 1+s, 1+s+p)| + |\mathsf{Misper}(V, 1+s, 1+s+p)| \le k$. Further, we have $|\mathsf{Misper}(V', 1+s+p, 1+s+2p)| \le |\mathsf{Misper}(V, 1+s, 1+s+p)|$. Thus, $|\mathsf{MP}(U, V')| \le |\mathsf{Misper}(U, 1+s, 1+s+p)| + |\mathsf{Misper}(V', 1+s, 1+s+p)| \le k$. This contradicts our assumption that $i^S + i^T$ was minimum possible. ◀

It remains to show that the sets A^S and A^T can be constructed efficiently. A τ-synchronising set can be computed in $\mathcal{O}(n)$ time by Theorem 7 and all the τ-runs, together with the position of the first occurrence of their Lyndon root, can be computed in $\mathcal{O}(n)$ time [8]. After an $\mathcal{O}(n)$-time preprocessing, for every τ-run, we can compute the set of the $k+1$ misperiods of its period to either side in $\mathcal{O}(1)$ time; see [29, Claim 18]. ◀

The next step in our solutions to long LCS and medium-length LCS was to construct an instance of the Two String Families LCP Problem. To adapt this approach, we generalise the notions of LCP and maxPairLCP so that they allow for mismatches. By $\mathsf{LCP}_k(U, V)$, for $k \in \mathbb{Z}_{\ge 0}$, we denote the maximum length ℓ such that $U[1\,..\,\ell]$ and $V[1\,..\,\ell]$ are at Hamming distance at most k.

▶ **Definition 26.** *Given two sets $\mathcal{U}, \mathcal{V} \subseteq \Sigma^* \times \Sigma^*$ and two integers $k_1, k_2 \in \mathbb{Z}_{\ge 0}$, we define* $\mathrm{maxPairLCP}_{k_1,k_2}(\mathcal{U}, \mathcal{V}) = \max\{\mathsf{LCP}_{k_1}(U_1, V_1) + \mathsf{LCP}_{k_2}(U_2, V_2) : (U_1, U_2) \in \mathcal{U}, (V_1, V_2) \in \mathcal{V}\}$.

Note that $\mathrm{maxPairLCP}(\mathcal{U}, \mathcal{V}) = \mathrm{maxPairLCP}_{0,0}(\mathcal{U}, \mathcal{V})$.

By Lemma 23, if a k-LCS of S and T has length $\ell' \in (\ell/2, \ell]$, then

$$\ell' = \max_{k'=0}^{k} \mathrm{maxPairLCP}_{k',k-k'}(\mathcal{U}, \mathcal{V}), \text{ for } \mathcal{U} = \{((S[a-\ell\,..\,a))^R, S[a\,..\,a+\ell)) : a \in A^S\},$$
$$\mathcal{V} = \{((T[a-\ell\,..\,a))^R, T[a\,..\,a+\ell)) : a \in A^T\}.$$

Here, k' bounds the number of mismatches between $S[i^S \mathinner{.\,.} i^S+\delta)$ and $T[i^T \mathinner{.\,.} i^T+\delta)$, whereas $k-k'$ bounds the number of mismatches between $S[i^S+\delta \mathinner{.\,.} i^S+\ell')$ and $T[i^T+\delta \mathinner{.\,.} i^T+\ell')$. The following theorem, whose full proof given in Section 6 is the most technical part of our contribution, allows for efficiently computing the values $\mathrm{maxPairLCP}_{k',k-k'}(\mathcal{U},\mathcal{V})$.

▶ **Theorem 27.** *Consider two (ℓ,ℓ)-families $\mathcal{U},\mathcal{V}$ of total size N consisting of pairs of substrings of a given length-n text. For any non-negative integers $k_1,k_2=\mathcal{O}(1)$, the value* $\mathrm{maxPairLCP}_{k_1,k_2}(\mathcal{U},\mathcal{V})$ *can be computed:*

- *in $\mathcal{O}(n+N\log^{k_1+k_2+1}N)$ time and $\mathcal{O}(n+N)$ space if $\ell>\log^{3/2}N$,*
- *in $\mathcal{O}(n+N\ell\log^{k_1+k_2-1/2}N)$ time and $\mathcal{O}(n+N\ell/\log N)$ space if $\log N<\ell\leq\log^{3/2}N$,*
- *in $\mathcal{O}(n+N\ell^{k_1+k_2}\sqrt{\log N})$ time and $\mathcal{O}(n+N)$ space if $\ell\leq\log N$.*

Proof Outline. We reduce the computation of $\mathrm{maxPairLCP}_{k_1,k_2}(\mathcal{U},\mathcal{V})$ into multiple computations of $\mathrm{maxPairLCP}(\mathcal{U}',\mathcal{V}')$ across a family $\mathbf{P}$ of pairs $(\mathcal{U}',\mathcal{V}')$ with $\mathcal{U}',\mathcal{V}'\subseteq\Sigma^*\times\Sigma^*$. Each pair $(U_1',U_2')\in\mathcal{U}'$ is associated to a pair $(U_1,U_2)\in\mathcal{U}$, with the string U_i' represented as a pointer to the *source* U_i and up to k_i substitutions needed to transform U_i to U_i'. Similarly, each pair $(V_1',V_2')\in\mathcal{V}'$ consists of *modified strings* with sources $(V_1,V_2)\in\mathcal{V}$. In order to guarantee $\mathrm{maxPairLCP}_{k_1,k_2}(\mathcal{U},\mathcal{V})=\max_{(\mathcal{U}',\mathcal{V}')\in\mathbf{P}}\mathrm{maxPairLCP}(\mathcal{U}',\mathcal{V}')$, we require $\mathsf{LCP}(U_i',V_i')\leq\mathsf{LCP}_{k_i}(U_i,V_i)$ for every $(U_1',U_2')\in\mathcal{U}'$ and $(V_1',V_2')\in\mathcal{V}'$ with $(\mathcal{U}',\mathcal{V}')\in\mathbf{P}$ and that, for every $(U_1,U_2)\in\mathcal{U}$ and $(V_1,V_2)\in\mathcal{V}$, there exists $(\mathcal{U}',\mathcal{V}')\in\mathbf{P}$ with $(U_1',U_2')\in\mathcal{U}'$ and $(V_1',V_2')\in\mathcal{V}'$, with sources (U_1,U_2) and (V_1,V_2), respectively, such that $\mathsf{LCP}(U_i',V_i')=\mathsf{LCP}_{k_i}(U_i,V_i)$. Our construction is based on a technique of [63] which gives an analogous family for two subsets of Σ^* (rather than $\Sigma^*\times\Sigma^*$) and a single threshold: We apply the approach of [63] to $\mathcal{U}_i=\{U_i:(U_1,U_2)\in\mathcal{U}\}$ and $\mathcal{V}_i=\{V_i:(V_1,V_2)\in\mathcal{V}\}$ with threshold k_i, and then combine the two resulting families $\mathbf{P}_i$ to derive $\mathbf{P}$.

Strengthening the arguments of [63], we show that each string $F_i\in\mathcal{U}_i\cup\mathcal{V}_i$ is the source of $\mathcal{O}(1)$ modified strings $F_i'\in\mathcal{U}_i'\cup\mathcal{V}_i'$ for any single $(\mathcal{U}_i',\mathcal{V}_i')\in\mathbf{P}_i$ and $\mathcal{O}(\min(\ell,\log N)^{k_i})$ modified strings across all $(\mathcal{U}_i',\mathcal{V}_i')\in\mathbf{P}_i$. This allows bounding the size of individual sets $(\mathcal{U}',\mathcal{V}')\in\mathbf{P}$ by $\mathcal{O}(N)$ and the overall size by $\mathcal{O}(N\min(\ell,\log N)^{k_1+k_2})$. In order to efficiently build the compacted tries required at the input of the Two String Families LCP Problem, the modified strings $F_i'\in\mathcal{U}_i'\cup\mathcal{V}_i'$ are sorted lexicographically, and the two derived linear orders (for $i\in\{1,2\}$) are maintained along with every pair $(\mathcal{U}',\mathcal{V}')\in\mathbf{P}$. Overall, the family $\mathbf{P}$ is constructed in $\mathcal{O}(n+N\min(\ell,\log N)^{k_1+k_2})$ time and $\mathcal{O}(n+N)$ space.

The resulting instances of the Two String Families LCP Problem are solved using Lemma 3 (if $\ell>\log^{3/2}N$) or Lemma 4 otherwise; note that $\mathcal{U}',\mathcal{V}'$ are (ℓ,ℓ)-families. ◀

Recall that the algorithm of Theorem 27 is called $k+1=\mathcal{O}(1)$ times, always with $N=|A^S|+|A^T|=\mathcal{O}(n/\ell)$. Overall, the value $\max_{k'=0}^{k}\mathrm{maxPairLCP}_{k',k-k'}(\mathcal{U},\mathcal{V})$ is therefore computed in $\mathcal{O}(n\log^{k-1/2}n)$ time and $\mathcal{O}(n)$ space in each of the following cases:

- in $\mathcal{O}(n+\frac{n}{\ell}\log^{k+1}N)=\mathcal{O}(n\log^{k-1/2}n)$ time and $\mathcal{O}(n+\frac{n}{\ell})=\mathcal{O}(n)$ space if $\ell>\log^{3/2}N$;
- in $\mathcal{O}(n+\frac{n}{\ell}\ell\log^{k-1/2}N)=\mathcal{O}(n\log^{k-1/2}n)$ time and $\mathcal{O}(n+\frac{n}{\ell}\ell/\log N)=\mathcal{O}(n)$ space if $\log N<\ell\leq\log^{3/2}N$;
- in $\mathcal{O}(n+\frac{n}{\ell}\ell^k\sqrt{\log N})=\mathcal{O}(n\log^{k-1}n)$ time and $\mathcal{O}(n+\frac{n}{\ell})=\mathcal{O}(n)$ space if $\ell\leq\log N$.

Accounting for $\mathcal{O}(n)$ time and space to determine the length d of an LCS between S and T, and the $\mathcal{O}(\log k)$ values ℓ that need to be tested so that that the intervals $(\ell/2,\ell]$ cover $[d,(k+1)d+k]$, this concludes the proof of Theorem 2.

Moreover, the three cases yield the following complexities: $\mathcal{O}(n+\frac{n}{\ell}\log^{k+1}n)$ if $\ell>\log^{3/2}N$, $\mathcal{O}(n\log^{k-1/2}n)=\mathcal{O}(\frac{n}{\ell}\log^{k+1}n)$ if $\log N<\ell\leq\log^{3/2}N$, and $\mathcal{O}(n\log^{k-1}n)=\mathcal{O}(\frac{n}{\ell}\log^{k+1}n)$ if $\ell\leq\log N$, which gives an $\mathcal{O}(n+\frac{n}{\ell}\log^{k+1}n)$-time solution for any ℓ, thus improving [26] for $k=\mathcal{O}(1)$.

6 Computing $\mathrm{maxPairLCP}_{k_1,k_2}(\mathcal{U},\mathcal{V})$—Proof of Theorem 27

For two strings $U, U' \in \Sigma^m$, we define the *mismatch positions* $\mathsf{MP}(U,U') = \{i \in [1,n] : U[i] \neq U'[i]\}$ and the *mismatch information* $\mathsf{MI}(U,U') = \{(i, U'[i]) : i \in \mathsf{MP}(U,U')\}$. Observe that U and $\mathsf{MI}(U,U')$ uniquely determine U'. This motivates the following definition:

▶ **Definition 28.** *Given $U \in \Sigma^m$ and $\Delta \subseteq [1,m] \times \Sigma$, we denote by U^Δ the unique string U' such that $\mathsf{MI}(U,U') = \Delta$. If there is no such string U', then U^Δ is undefined. We say that U^Δ, represented using a pointer to U and the set Δ, is a* modified string *with* source *U.*

▶ **Example 29.** Let $U = \mathtt{abababb}$ and $\Delta = \{(2,\mathtt{a}),(3,\mathtt{b})\}$. Then $U^\Delta = U' = \mathtt{aabbbab}$.

▶ **Definition 30** (see [63]). *Given strings $U, V \in \Sigma^*$ and an integer $k \in \mathbb{Z}_{\geq 0}$, we say that two modified strings (U^Δ, V^∇) form a* $(U,V)_k$-maxpair *if the following holds for every i:*

- *if $i \in [1, \mathsf{LCP}_k(U,V)]$ and $U[i] \neq V[i]$, then $U^\Delta[i] = V^\nabla[i]$*
- *otherwise, $U^\Delta[i] = U[i]$ (assuming $i \in [1,|U|]$) and $V^\nabla[i] = V[i]$ (assuming $i \in [1,|V|]$).*

▶ **Example 31.** Let $U = \mathtt{ababbabb}$, $V = \mathtt{aacbaaab}$ and $k = 3$. Further let $\Delta = \{(2,\mathtt{a}),(3,\mathtt{b})\}$ and $\nabla = \{(3,\mathtt{b}),(5,\mathtt{b}))\}$. We have $U^\Delta = \mathtt{aabbbabb}$ and $U^\nabla = \mathtt{aabbbaab}$. Then $\mathsf{LCP}_k(U,V) = 6$ and $(U^\Delta, V^\nabla) = (\mathtt{aabbbaab}, \mathtt{aabbbabb})$ form a $(U,V)_3$-maxpair.

The following simple fact characterises this notion.

▶ **Fact 32.** *Let U^Δ, V^∇ be modified strings with sources $U, V \in \Sigma^*$ and let $k \in \mathbb{Z}_{\geq 0}$.*
(a) *If (U^Δ, V^∇) is a $(U,V)_k$-maxpair, then $|\Delta \cup \nabla| \leq k$ and $\mathsf{LCP}(U^\Delta, V^\nabla) \geq \mathsf{LCP}_k(U,V)$.*
(b) *If $|\Delta \cup \nabla| \leq k$, then $\mathsf{LCP}_k(U,V) \geq \mathsf{LCP}(U^\Delta, V^\nabla)$.*

Proof. (a) Let $d = \mathsf{LCP}_k(U,V)$ and $M = \mathsf{MP}(U[1\,..\,d], V[1\,..\,d])$. By Definition 30, we have $U^\Delta[i] = V^\nabla[i]$ for $i \in [1,d]$. Consequently, $\mathsf{LCP}(U^\Delta, V^\nabla) \geq d$ and, if $(i,a) \in \Delta$ and $(i,b) \in \nabla$ holds for some $i \in [1,d]$, then $a = b$. Furthermore, Definition 30 yields $\Delta, \nabla \subseteq M \times \Sigma$, and hence $|\Delta \cup \nabla| \leq |M| \leq k$.
(b) Let $d' = \mathsf{LCP}(U^\Delta, V^\nabla)$ and $M' = \mathsf{MP}(U[1\,..\,d'], V[1\,..\,d'])$. For every $i \in M'$, we have $U[i] \neq V[i]$ yet $U^\Delta[i] = V^\nabla[i]$. This implies $U[i] \neq U^\Delta[i]$ or $V[i] \neq V^\nabla[i]$, i.e., $(i, U^\Delta[i]) = (i, V^\nabla[i]) \in \Delta \cup \nabla$. Consequently, $|M'| \leq |\Delta \cup \nabla| \leq k$, which means that $\mathsf{LCP}_k(U,V) \geq d'$ holds as claimed. ◀

▶ **Definition 33.** *Consider a set of strings $\mathcal{F} \subseteq \Sigma^*$ and an integer $k \in \mathbb{Z}_{\geq 0}$. A k-*complete family *for $\mathcal{F}$ is a family $\mathbf{F}$ of sets of modified strings of the form F^Δ for $F \in \mathcal{F}$ and $|\Delta| \leq k$ such that, for every $U, V \in \mathcal{F}$, there exists a set $\mathcal{F}' \in \mathbf{F}$ and modified strings $U^\Delta, V^\nabla \in \mathcal{F}'$ forming a $(U,V)_k$-maxpair.*

▶ **Example 34.** Let $\mathcal{U}$ and $\mathcal{V}$ be the sets of all suffixes of strings S and T, respectively, and $\mathbf{F}$ be a k-complete family for $\mathcal{U} \cup \mathcal{V}$. Then

$$k\text{-LCS}(S,T) = \max_{\mathcal{F}' \in \mathbf{F}} \{\mathsf{LCP}(U^\Delta, V^\nabla) : U \in \mathcal{U},\, V \in \mathcal{V},\, |\Delta \cup \nabla| \leq k,\, U^\Delta, V^\nabla \in \mathcal{F}'\}.$$

Our construction of a k-complete family follows the approach of Thankachan et al. [63] (which, in turn, is based on the ideas behind k-errata trees of [32]) with minor modifications. For completeness, we provide a full proof of the following proposition in Appendix A.

▶ **Proposition 35** (see [63]). *Let $\mathcal{F} \subseteq \Sigma^{\leq \ell}$ and $k \in \mathbb{Z}_{\geq 0}$ with $k = \mathcal{O}(1)$. There exists a k-complete family $\mathbf{F}$ for $\mathcal{F}$ such that, for each $F \in \mathcal{F}$:*

- *Every individual set $\mathcal{F}' \in \mathbf{F}$ contains $\mathcal{O}(1)$ modified strings with source F.*

- *In total, the sets $\mathcal{F}' \in \mathbf{F}$ contain $\mathcal{O}(\min(\ell, \log|\mathcal{F}|)^k)$ modified strings with source F.*

Moreover, if $\mathcal{F}$ consists of substrings of a given length-n text, then the family $\mathbf{F}$ can be constructed in $\mathcal{O}(n + |\mathcal{F}|)$ space and $\mathcal{O}(n + |\mathcal{F}| \min(\ell, \log|\mathcal{F}|)^k)$ time with sets $\mathcal{F}' \in \mathbf{F}$ generated one by one and modified strings within each set $\mathcal{F}' \in \mathbf{F}$ sorted lexicographically.

Intuitively, in the approach of Thankachan et al. [63] a k-LCS was computed as the maximum LCP_k of any two suffixes originating from different strings S, T. Hence, using the k-complete family shown in Example 34 was sufficient. However, in our approach, a k-LCS is anchored at some pair of synchronised positions. This motivates the following generalised notion aimed to account for the parts of a k-LCS on both sides of the anchor.

► **Definition 36.** *Consider a set $\mathcal{G} \subseteq \Sigma^* \times \Sigma^*$ of string pairs and integers $k_1, k_2 \in \mathbb{Z}_{\geq 0}$. A (k_1, k_2)-*bicomplete family *for $\mathcal{G}$ is a family $\mathbf{G}$ of sets $\mathcal{G}'$ of modified string pairs of the form $(F_1^{\Delta_1}, F_2^{\Delta_2})$ for $(F_1, F_2) \in \mathcal{G}$, $|\Delta_1| \leq k_1$, and $|\Delta_2| \leq k_2$, such that, for every $(U_1, U_2), (V_1, V_2) \in \mathcal{G}$, there exists a set $\mathcal{G}' \in \mathbf{G}$ with $(U_1^{\Delta_1}, U_2^{\Delta_2}), (V_1^{\nabla_1}, V_2^{\nabla_2}) \in \mathcal{G}'$ such that $(U_1^{\Delta_1}, V_1^{\nabla_1})$ is a $(U_1, V_1)_{k_1}$-maxpair and $(U_2^{\Delta_2}, V_2^{\nabla_2})$ is a $(U_2, V_2)_{k_2}$-maxpair.*

► **Example 37.** Let $\mathcal{U}$ and $\mathcal{V}$ be the sets of string pairs and $\mathbf{G}$ be a (k_1, k_2)-bicomplete family for $\mathcal{U} \cup \mathcal{V}$. Then

$$\begin{aligned}&\mathrm{maxPairLCP}_{k_1,k_2}(\mathcal{U}, \mathcal{V}) = \max_{\mathcal{G}' \in \mathbf{G}} \{\mathsf{LCP}(U_1^{\Delta_1}, V_1^{\nabla_1}) + \mathsf{LCP}(U_2^{\Delta_2}, V_2^{\nabla_2}) :\\ &(U_1, U_2) \in \mathcal{U}, (V_1, V_2) \in \mathcal{V}, |\Delta_1 \cup \nabla_1| \leq k_1, |\Delta_2 \cup \nabla_2| \leq k_2,\ (U_1^{\Delta_1}, U_2^{\Delta_2}), (V_1^{\nabla_1}, V_2^{\nabla_2}) \in \mathcal{G}'\}.\end{aligned}$$

Complete families can be used to efficiently construct a bicomplete family as shown in the following lemma. The construction requires proper care to ensure that only linear space is used.

► **Lemma 38.** *Let $\mathcal{G}$ be an (ℓ, ℓ)-family and $k_1, k_2 \in \mathbb{Z}_{\geq 0}$ with $k_1, k_2 = \mathcal{O}(1)$. There exists a (k_1, k_2)-bicomplete family $\mathbf{G}$ for $\mathcal{G}$ such that, for each $(F_1, F_2) \in \mathcal{G}$:*

- *Every individual set $\mathcal{G}' \in \mathbf{G}$ contains $\mathcal{O}(1)$ pairs of the form $(F_1^{\Delta_1}, F_2^{\Delta_2})$;*
- *In total, the sets $\mathcal{G}' \in \mathbf{G}$ contain $\mathcal{O}(\min(\ell, \log|\mathcal{G}|)^{k_1+k_2})$ pairs of the form $(F_1^{\Delta_1}, F_2^{\Delta_2})$.*

Moreover, if $\mathcal{G}$ consists of pairs of substrings of a given length-n text, then the family $\mathbf{G}$ can be constructed in $\mathcal{O}(n + |\mathcal{G}|)$ space and $\mathcal{O}(n + |\mathcal{G}| \min(\ell, \log|\mathcal{G}|)^{k_1+k_2})$ time with sets $\mathcal{G}' \in \mathbf{G}$ generated one by one and each set $\mathcal{G}'$ sorted in two ways: according to the lexicographic order of the modified strings on the first and the second coordinate, respectively.

Proof. Let $\mathcal{F}_1 = \{F_1 : (F_1, F_2) \in \mathcal{G}\}$ and $\mathcal{F}_2 = \{F_2 : (F_1, F_2) \in \mathcal{G}\}$. Moreover, for $i \in \{1, 2\}$, let $\mathbf{F}_i$ be the k_i-complete family for $\mathcal{F}_i$ obtained using Proposition 35. The family $\mathbf{G}$ is defined as follows:

$$\mathbf{G} = \{\{(F_1^{\Delta_1}, F_2^{\Delta_2}) \in \mathcal{F}'_1 \times \mathcal{F}'_2 : (F_1, F_2) \in \mathcal{G}\} : (\mathcal{F}'_1, \mathcal{F}'_2) \in \mathbf{F}_1 \times \mathbf{F}_2\} \setminus \{\emptyset\}.$$

Clearly, each set $\mathcal{G}' \in \mathbf{G}$ consists of modified string pairs of the form $(F_1^{\Delta_1}, F_2^{\Delta_2})$ with $(F_1, F_2) \in \mathcal{G}$, $|\Delta_1| \leq k_1$, and $|\Delta_2| \leq k_2$. Let us fix pairs $(U_1, U_2), (V_1, V_2) \in \mathcal{G}$. By Definition 33, for $i \in \{1, 2\}$, there exists a set $\mathcal{F}'_i \in \mathbf{F}_i$ and modified strings $U_i^{\Delta_i}, V_i^{\nabla_i} \in \mathcal{F}'_i$ that form a $(U_i, V_i)_{k_i}$-maxpair. The family $\mathbf{G}$ contains a set $\mathcal{G}' = \{(F_1^{\Delta_1}, F_2^{\Delta_2}) \in \mathcal{F}'_1 \times \mathcal{F}'_2 : (F_1, F_2) \in \mathcal{G}\}$ and this set $\mathcal{G}'$ contains both $(U_1^{\Delta_1}, U_2^{\Delta_2})$ and $(V_1^{\nabla_1}, V_2^{\nabla_2})$. Thus, $\mathbf{G}$ is a (k_1, k_2)-bicomplete family for $\mathcal{G}$.

Now, let us fix a pair $(F_1, F_2) \in \mathcal{G}$ in order to bound the number of modified string pairs of the form $(F_1^{\Delta_1}, F_2^{\Delta_2})$ contained in the sets $\mathcal{G}' \in \mathbf{G}$. Observe that if $(F_1^{\Delta_1}, F_2^{\Delta_2}) \in \mathcal{G}'$, where $\mathcal{G}'$ is constructed for $(\mathcal{F}'_1, \mathcal{F}'_2) \in \mathbf{F}_1 \times \mathbf{F}_2$, then $F_i^{\Delta_i} \in \mathcal{F}'_i$. By Proposition 35, for $i \in \{1,2\}$, the set $\mathcal{F}'_i \in \mathbf{F}_i$ contains $\mathcal{O}(1)$ modified strings of the form $F_i^{\Delta_i}$. Consequently, the set $\mathcal{G}' \in \mathbf{G}$ contains $\mathcal{O}(1)$ modified string pairs of the form $(F_1^{\Delta_1}, F_2^{\Delta_2})$. Moreover, Proposition 35 implies that, for $i \in \{1,2\}$, the sets $\mathcal{F}'_i \in \mathbf{F}_i$ in total contain $\mathcal{O}(\min(\ell, \log|\mathcal{F}_i|)^{k_i}) = \mathcal{O}(\min(\ell, \log|\mathcal{G}|)^{k_i})$ modified strings of the form $F_i^{\Delta_i}$. Thus, the sets $\mathcal{G}' \in \mathbf{G}$ in total contain $\mathcal{O}(\min(\ell, \log|\mathcal{G}|)^{k_1+k_2})$ modified string pairs of the form $(F_1^{\Delta_1}, F_2^{\Delta_2})$.

It remains to describe an efficient algorithm constructing the family $\mathbf{G}$. We first generate the family $\mathbf{F}_1$ and batch it into subfamilies $\mathbf{F}'_1 \subseteq \mathbf{F}_1$ using the following procedure applied on top of the algorithm of Proposition 35 after it outputs a set $\mathcal{F}'_1 \in \mathbf{F}_1$. For each modified string $F_1^{\Delta_1} \in \mathcal{F}'_1$, we augment each pair of the form $(F_1, F_2) \in \mathcal{G}$ with a triple consisting of $F_1^{\Delta_1}$, the lexicographic rank of $F_1^{\Delta_1}$ within $\mathcal{F}'_1$, and the index of $\mathcal{F}'_1$ within $\mathbf{F}'_1$. After processing a set $\mathcal{F}'_1 \in \mathbf{F}_1$ in this manner, we resume the algorithm of Proposition 35 provided that the total number of triples stored is smaller than $n + |\mathcal{G}|$. Otherwise (and when the algorithm of Proposition 35 terminates), we declare the construction of the current batch $\mathbf{F}'_1 \subseteq \mathbf{F}_1$ complete.

For each batch $\mathbf{F}'_1 \subseteq \mathbf{F}_1$, we generate the family $\mathbf{F}_2$ and batch it into subfamilies $\mathbf{F}'_2 \subseteq \mathbf{F}_2$ using the following procedure applied on top of the algorithm of Proposition 35 after it outputs a set $\mathcal{F}'_2 \in \mathbf{F}_2$. For each modified string $F_2^{\Delta_2} \in \mathcal{F}'_2$, we retrieve all pairs of the form $(F_1, F_2) \in \mathcal{G}$ and iterate over the triples stored at (F_1, F_2). For each such triple, consisting of a modified string $F_1^{\Delta_1}$, the lexicographic rank of $F_1^{\Delta_1}$ within $\mathcal{F}'_1 \in \mathbf{F}'_1$, and the index of $\mathcal{F}'_1$ within $\mathbf{F}'_1$, we generate the corresponding triple for $F_2^{\Delta_2}$ and combine the two triples into a 6-tuple. After processing a set $\mathcal{F}'_2 \in \mathbf{F}_2$ in this manner, we resume the algorithm of Proposition 35 provided that the total number of 6-tuples stored is smaller than $n + |\mathcal{G}|$. Otherwise (and when the algorithm of Proposition 35 terminates), we declare the construction of the current batch $\mathbf{F}'_2 \subseteq \mathbf{F}_2$ complete.

For each pair of batches $\mathbf{F}'_1, \mathbf{F}'_2$, we group the 6-tuples by the index of $\mathcal{F}'_1$ within $\mathbf{F}'_1$ and the index of $\mathcal{F}'_2$ within $\mathbf{F}'_2$, and we sort the 6-tuples in each group in two ways: by the lexicographic rank of $F_1^{\Delta_1}$ within $\mathcal{F}'_1$ and by the lexicographic rank of $F_2^{\Delta_2}$ within $\mathcal{F}'_2$. The keys used for sorting and grouping are integers bounded by $n^{\mathcal{O}(1)}$, so we implement this step using radix sort. Finally, for each group, we create a set $\mathcal{G}'$ by preserving only the modified strings $(F_1^{\Delta_1}, F_2^{\Delta_2})$ out of each 6-tuple. We output $\mathcal{G}'$ along with the two linear orders: according to $F_1^{\Delta_1}$ and $F_2^{\Delta_2}$. It is easy to see that this yields $\mathcal{G}' = \{(F_1^{\Delta_1}, F_2^{\Delta_2}) \in \mathcal{F}'_1 \times \mathcal{F}'_2 : (F_1, F_2) \in \mathcal{G}\}$. Consequently, for the two batches $\mathbf{F}'_1, \mathbf{F}'_2$, the algorithm produces

$$\mathbf{G}' = \{\{(F_1^{\Delta_1}, F_2^{\Delta_2}) \in \mathcal{F}'_1 \times \mathcal{F}'_2 : (F_1, F_2) \in \mathcal{G}\} : (\mathcal{F}'_1, \mathcal{F}'_2) \in \mathbf{F}'_1 \times \mathbf{F}'_2\} \setminus \{\emptyset\}.$$

Since, for $i \in \{1,2\}$, each set $\mathcal{F}'_i \in \mathbf{F}_i$ belongs to exactly one batch $\mathbf{F}'_i$, across all pairs of batches we obtain the family $\mathbf{G}$ defined above.

We conclude with the complexity analysis. The generators of $\mathbf{F}_1$ and $\mathbf{F}_2$ use $\mathcal{O}(n+|\mathcal{F}_1|) = \mathcal{O}(n+|\mathcal{G}|)$ and $\mathcal{O}(n+|\mathcal{F}_2|) = \mathcal{O}(n+|\mathcal{G}|)$ space, respectively. Each set $\mathcal{F}'_1 \in \mathbf{F}_1$ contains $\mathcal{O}(1)$ modified strings with the same source, so the number of triples generated for $\mathcal{F}'_1$ is $\mathcal{O}(|\mathcal{G}|)$, so the number of triples generated for each batch $\mathbf{F}'_1$ is $\mathcal{O}(n+|\mathcal{G}|)$. Similarly, each set $\mathcal{F}'_2 \in \mathbf{F}_2$ contains $\mathcal{O}(1)$ modified strings with the same source, so the number of 6-tuples generated for each batch pair $\mathbf{F}'_1, \mathbf{F}'_2$ is $\mathcal{O}(n+|\mathcal{G}|)$. The triples are removed after processing each batch $\mathbf{F}'_1$ and the 6-tuples are removed after processing each pair of batches $\mathbf{F}'_1, \mathbf{F}'_2$, so the space complexity of the entire algorithm is $\mathcal{O}(n+|\mathcal{G}|)$.

As for the running time, note that the generator of $\mathbf{F}_1$ takes $\mathcal{O}(n+|\mathcal{F}_1|\min(\ell, \log|\mathcal{F}_1|)^{k_1}) = \mathcal{O}(n+|\mathcal{G}|\min(\ell, \log|\mathcal{G}|)^{k_1})$ time. In the post-processing of the sets $\mathcal{F}'_1 \in \mathbf{F}_1$, we generate $\mathcal{O}(|\mathcal{G}|\min(\ell, \log|\mathcal{F}_1|)^{k_1}) = \mathcal{O}(|\mathcal{G}|\min(\ell, \log|\mathcal{G}|)^{k_1})$ triples in $\mathcal{O}(1)$ time per triple. The number of batches $\mathbf{F}'_1$ is therefore $\mathcal{O}(1 + \frac{|\mathcal{G}|}{n+|\mathcal{G}|}\min(\ell, \log|\mathcal{G}|)^{k_1})$. For each such batch, we run the generator of $\mathbf{F}_2$, which takes $\mathcal{O}(n+|\mathcal{F}_2|\min(\ell,\log|\mathcal{F}_2|)^{k_2}) = \mathcal{O}(n+|\mathcal{G}|\min(\ell,\log|\mathcal{G}|)^{k_2})$ time. Across all batches $\mathbf{F}'_1$, this sums up to $\mathcal{O}((1+\frac{|\mathcal{G}|}{n+|\mathcal{G}|}\min(\ell,\log|\mathcal{G}|)^{k_1})(n+|\mathcal{G}|\min(\ell,\log|\mathcal{G}|)^{k_2})) = \mathcal{O}(n+|\mathcal{G}|\min(\ell,\log|\mathcal{G}|)^{k_1+k_2})$. In the post-processing of the sets $\mathcal{F}'_2 \in \mathbf{F}_2$, we generate $\sum_{\mathcal{G}'\in\mathbf{G}}|\mathcal{G}'| = \mathcal{O}(|\mathcal{G}|\min(\ell,\log|\mathcal{G}|)^{k_1+k_2})$ 6-tuples, in $\mathcal{O}(1)$ time per tuple. The number of batch pairs $\mathbf{F}'_1, \mathbf{F}'_2$ is therefore $\mathcal{O}(1+\frac{|\mathcal{G}|}{n+|\mathcal{G}|}\min(\ell,\log|\mathcal{G}|)^{k_1}+\frac{|\mathcal{G}|}{n+|\mathcal{G}|}\min(\ell,\log|\mathcal{G}|)^{k_1+k_2}) = \mathcal{O}(1+\frac{|\mathcal{G}|}{n+|\mathcal{G}|}\min(\ell,\log|\mathcal{G}|)^{k_1+k_2})$. Each batch pair is processed in $\mathcal{O}(n+|\mathcal{G}|)$ time, so this yields $\mathcal{O}(n+|\mathcal{G}|\min(\ell,\log|\mathcal{G}|)^{k_1+k_2})$ time in total. ◀

Next we will use bicomplete families to reduce the computation of $\text{maxPairLCP}_{k_1,k_2}(\mathcal{U},\mathcal{V})$ for (ℓ,ℓ)-families of substrings of a given string to a number of computations of $\text{maxPairLCP}(\mathcal{U}',\mathcal{V}')$ for (ℓ,ℓ)-families of modified substrings. Similarly to [63], we need to group the elements of a bicomplete family for $\mathcal{U}\cup\mathcal{V}$ by subsets of modifications so that, if two modified strings have a common modification, it is counted as one mismatch between them. Intuitively, we primarily want to avoid checking the conditions $|\Delta_i \cup \nabla_i| \le k_i$ for $i \in \{1,2\}$ in the formula in Example 37.

▶ **Proposition 39.** *Let $\mathcal{U},\mathcal{V}$ be (ℓ,ℓ)-families and $k_1,k_2 \in \mathbb{Z}_{\ge 0}$ with $k_1,k_2 = \mathcal{O}(1)$. There exists a family $\mathbf{P}$ such that:*

1. *each element of $\mathbf{P}$ is a pair of sets $(\mathcal{U}',\mathcal{V}')$, where the elements of $\mathcal{U}'$ are of the form $(U_1^{\Delta_1}, U_2^{\Delta_2})$ for $(U_1,U_2)\in\mathcal{U}$ with $|\Delta_1|\le k_1$ and $|\Delta_2|\le k_2$, whereas the elements of $\mathcal{V}'$ are of the form $(V_1^{\nabla_1}, V_2^{\nabla_2})$ for $(V_1,V_2)\in\mathcal{V}$ with $|\nabla_1|\le k_1$ and $|\nabla_2|\le k_2$;*
2. *$\max_{(\mathcal{U}',\mathcal{V}')\in\mathbf{P}}(|\mathcal{U}'|+|\mathcal{V}'|) = \mathcal{O}(|\mathcal{U}|+|\mathcal{V}|)$;*
3. *$\sum_{(\mathcal{U}',\mathcal{V}')\in\mathbf{P}}(|\mathcal{U}'|+|\mathcal{V}'|) = \mathcal{O}((|\mathcal{U}|+|\mathcal{V}|)\min(\ell,\log(|\mathcal{U}|+|\mathcal{V}|))^{k_1+k_2})$;*
4. *$\max_{(\mathcal{U}',\mathcal{V}')\in\mathbf{P}}\text{maxPairLCP}(\mathcal{U}',\mathcal{V}') = \text{maxPairLCP}_{k_1,k_2}(\mathcal{U},\mathcal{V})$.*

Moreover, if $\mathcal{U}$ and $\mathcal{V}$ consist of pairs of substrings of a given length-n text, then the family $\mathbf{P}$ can be constructed in $\mathcal{O}(n+(|\mathcal{U}|+|\mathcal{V}|)\min(\ell,\log(|\mathcal{U}|+|\mathcal{V}|))^{k_1+k_2})$ time and $\mathcal{O}(n+|\mathcal{U}|+|\mathcal{V}|)$ space, with pairs $(\mathcal{U}',\mathcal{V}')$ generated one by one and each set $\mathcal{U}'\cup\mathcal{V}'$ sorted in two ways: according to the lexicographic order of the modified strings on the first and the second coordinate, respectively.

Proof. Let $\mathcal{G} = \mathcal{U}\cup\mathcal{V}$ and let $\mathbf{G}$ be the (k_1,k_2)-bicomplete family for $\mathcal{G}$. Given $\mathcal{G}'\in\mathbf{G}$ and $\delta_1,\delta_2 \subseteq \mathbb{Z}_{\ge 0}\times\Sigma$, we define

$$\mathcal{G}'_{\delta_1,\delta_2} = \{(F_1^{\Delta_1}, F_2^{\Delta_2}) \in \mathcal{G}' : \delta_1 \subseteq \Delta_1, \delta_2 \subseteq \Delta_2\}.$$

For all non-empty sets $\mathcal{G}'_{\delta_1,\delta_2}$ with $\mathcal{G}'\in\mathbf{G}$, and all integers $d_1\in[0,k_1]$ and $d_2\in[0,k_2]$, we insert to $\mathbf{P}$ a pair $(\mathcal{U}',\mathcal{V}')$, where

$$\mathcal{U}' = \{(U_1^{\Delta_1}, U_2^{\Delta_2}) \in \mathcal{G}'_{\delta_1,\delta_2} : (U_1,U_2)\in\mathcal{U},\ |\Delta_1|\le d_1,\ |\Delta_2|\le d_2\},$$
$$\mathcal{V}' = \{(V_1^{\nabla_1}, V_2^{\nabla_2}) \in \mathcal{G}'_{\delta_1,\delta_2} : (V_1,V_2)\in\mathcal{V},\ |\nabla_1|\le k_1+|\delta_1|-d_1,\ |\nabla_2|\le k_2+|\delta_2|-d_2\}.$$

Clearly, the elements of $\mathcal{U}'$ and $\mathcal{V}'$ satisfy requirement 1. Moreover, $|\mathcal{U}'|+|\mathcal{V}'| \le 2|\mathcal{G}'| = \mathcal{O}(|\mathcal{G}|) = \mathcal{O}(|\mathcal{U}|+|\mathcal{V}|)$, so requirement 2 is fulfilled. Furthermore, each pair $(F_1^{\Delta_1}, F_2^{\Delta_2}) \in \mathcal{G}'$ belongs to $2^{|\Delta_1|+|\Delta_2|} \le 2^{k_1+k_2} = \mathcal{O}(1)$ sets $\mathcal{G}'_{\delta_1,\delta_2}$ and thus to $\mathcal{O}(k_1k_2) = \mathcal{O}(1)$ sets $\mathcal{U}'$ or $\mathcal{V}'$ created for $\mathcal{G}'$. Consequently, due to $\sum_{\mathcal{G}'\in\mathbf{G}}|\mathcal{G}'| = \mathcal{O}(|\mathcal{G}|\min(\ell,\log|\mathcal{G}|)^{k_1+k_2})$, we have $\sum_{(\mathcal{U}',\mathcal{V}')\in\mathbf{P}}(|\mathcal{U}'|+|\mathcal{V}'|) = \mathcal{O}(|\mathcal{G}|\min(\ell,\log|\mathcal{G}|)^{k_1+k_2}) = \mathcal{O}((|\mathcal{U}|+|\mathcal{V}|)\min(\ell,\log(|\mathcal{U}|+|\mathcal{V}|))^{k_1+k_2})$, which yields requirement 3.

Our next goal is to prove $\text{maxPairLCP}_{k_1,k_2}(\mathcal{U},\mathcal{V}) = \max_{(\mathcal{U}',\mathcal{V}')\in\mathbf{P}}\text{maxPairLCP}(\mathcal{U}',\mathcal{V}')$ (requirement 4).

▶ **Example 40.** As in Example 31, consider $U = \texttt{ababbabb}$, $V = \texttt{aacbaaab}$, $k = 3$, $\Delta = \{(2, \texttt{a}), (3, \texttt{b})\}$, and $\nabla = \{(3, \texttt{b}), (5, \texttt{b})\}$. Recall that (U^Δ, V^∇) form a $(U, V)_3$-maxpair. We have $\delta = \Delta \cap \nabla = \{(3, \texttt{b})\}$. U^Δ has $d = 2$ modifications, and V^∇ has $k + |\delta| - d = 2$ modifications. Note that, if we instead had $\nabla = \{(3, \texttt{b}), (5, \texttt{b}), (7, \texttt{b})\}$, then $\mathsf{LCP}(U^\Delta, V^\nabla)$ would be greater than $\mathsf{LCP}_3(U, V)$, and we would thus not want to consider the pair (U^Δ, V^∇).

Let us fix $(\mathcal{U}', \mathcal{V}') \in \mathbf{P}$ generated for $\mathcal{G}'$, δ_1, δ_2, d_1, and d_2. For every $(U_1^{\Delta_1}, U_2^{\Delta_2}) \in \mathcal{U}'$ and $(V_1^{\nabla_1}, V_2^{\nabla_2}) \in \mathcal{V}'$, we have $|\Delta_i \cup \nabla_i| = |\Delta_i| + |\nabla_i| - |\Delta_i \cap \nabla_i| \le d_i + (k_i + |\delta_i| - d_i) - |\delta_i| \le k_i$ for $i \in \{1, 2\}$ and thus, by Fact 32(b), $\mathsf{LCP}(U_i^{\Delta_i}, V_i^{\nabla_i}) \le \mathsf{LCP}_{k_i}(U_i, V_i)$. Consequently, we have $\mathsf{LCP}(U_1^{\Delta_1}, V_1^{\nabla_1}) + \mathsf{LCP}(U_2^{\Delta_2}, V_2^{\nabla_2}) \le \mathsf{LCP}_{k_1}(U_1, V_1) + \mathsf{LCP}_{k_2}(U_2, V_2)$, and thus $\mathrm{maxPairLCP}(\mathcal{U}', \mathcal{V}') \le \mathrm{maxPairLCP}_{k_1,k_2}(\mathcal{U}, \mathcal{V})$.

As for the converse inequality, suppose that $(U_1, U_2) \in \mathcal{U}$ and $(V_1, V_2) \in \mathcal{V}$ satisfy $\mathrm{maxPairLCP}_{k_1,k_2}(\mathcal{U}, \mathcal{V}) = \mathsf{LCP}_{k_1}(U_1, V_1) + \mathsf{LCP}_{k_2}(U_2, V_2)$. By the definition of a bicomplete family (Definition 36), there exists $\mathcal{G}' \in \mathcal{G}$ and $(U_1^{\Delta_1}, U_2^{\Delta_2}), (V_1^{\nabla_1}, V_2^{\nabla_2}) \in \mathcal{G}'$ such that $(U_i^{\Delta_i}, V_i^{\nabla_i})$ is a $(U_i, V_i)_{k_i}$-maxpair for $i \in \{1, 2\}$. By Fact 32(a), this yields $\mathsf{LCP}_{k_i}(U_i, V_i) \le \mathsf{LCP}(U_i^{\Delta_i}, V_i^{\nabla_i})$ and $|\Delta_i \cup \nabla_i| \le k_i$. Define $\delta_i = \Delta_i \cap \nabla_i$ and $d_i = |\Delta_i|$, and consider the pair $(\mathcal{U}', \mathcal{V}')$ constructed for $\mathcal{G}'$, δ_1, δ_2, d_1, and d_2. Note that $|\Delta_i| \le d_i$ and $\delta_i \subseteq \Delta_i$, so $(U_1^{\Delta_1}, U_2^{\Delta_2}) \in \mathcal{U}'$. Moreover, $|\nabla_i| = |\Delta_i \cup \nabla_i| + |\delta_i| - |\Delta_i| \le k_i + |\delta_i| - d_i$ and $\delta_i \subseteq \nabla_i$, so $(V_1^{\nabla_1}, V_2^{\nabla_2}) \in \mathcal{V}'$. Consequently, $\mathrm{maxPairLCP}_{k_1,k_2}(\mathcal{U}, \mathcal{V}) \le \mathrm{maxPairLCP}(\mathcal{U}', \mathcal{V}')$.

Finally, we need to design an efficient algorithm constructing the family $\mathbf{P}$. We apply the construction of a bicomplete family (Lemma 38) to generate the family $\mathbf{G}$ in batches $\mathbf{G}'$ consisting of sets of total size $\Theta(n + |\mathcal{G}|)$ (the last batch might be smaller) [2]. For each batch $\mathbf{G}' \subseteq \mathbf{G}$, we first construct all the non-empty sets $\mathcal{G}'_{\delta_1,\delta_2}$ for $\mathcal{G}' \in \mathbf{G}'$. For this, we iterate over sets $\mathcal{G}' \in \mathbf{G}'$, pairs $(F_1^{\Delta_1}, F_2^{\Delta_2}) \in \mathcal{G}'$, subsets $\delta_1 \subseteq \Delta_1$, and subsets $\delta_2 \subseteq \Delta_2$, creating 5-tuples consisting of the index of $\mathcal{G}'$ in $\mathbf{G}'$, as well as δ_1, δ_2, $F_1^{\Delta_1}$, and $F_2^{\Delta_2}$. We group these 5-tuples according to the first three coordinates; the key consists of $\mathcal{O}(1 + k_1 + k_2) = \mathcal{O}(1)$ integers bounded by $n^{\mathcal{O}(1)}$, so we use radix sort. Moreover, since radix sort is stable, the sets $\mathcal{G}'_{\delta_1,\delta_2}$ can be constructed along with both linear orders derived from $\mathcal{G}'$. Finally, for each non-empty set $\mathcal{G}'_{\delta_1,\delta_2}$ with $\mathcal{G}' \in \mathbf{G}'$, we iterate over $d_1 \in [0, k_1]$ and $d_2 \in [0, k_2]$, generating the subsets $\mathcal{U}'$ and $\mathcal{V}'$ of $\mathcal{G}'_{\delta_1,\delta_2}$ according to the formulae above. The two linear orders of $\mathcal{U}' \cup \mathcal{V}'$ are derived from $\mathcal{G}'_{\delta_1,\delta_2}$.

We conclude with the complexity analysis. The algorithm of Lemma 38 takes $\mathcal{O}(n + |\mathcal{G}| \min(\ell, \log |\mathcal{G}|)^{k_1+k_2})$ time and $\mathcal{O}(n + |\mathcal{G}|)$ space. The total size of the sets in each batch $\mathbf{G}'$ is $\mathcal{O}(n + |\mathcal{G}|)$ and the number of batches is $\mathcal{O}(1 + \frac{|\mathcal{G}|}{n+|\mathcal{G}|} \min(\ell, \log |\mathcal{G}|)^{k_1+k_2})$. For each modified string $(F_1^{\Delta_1}, F_2^{\Delta_2}) \in \mathcal{G}'$, we construct $2^{|\Delta_1|+|\Delta_2|} \le 2^{k_1+k_2} = \mathcal{O}(1)$ tuples corresponding to elements of the sets $\mathcal{G}'_{\delta_1,\delta_2}$, in $\mathcal{O}(1)$ time per tuple. Consequently, this phase uses $\mathcal{O}(n + |\mathcal{G}|)$ space and $\mathcal{O}(|\mathcal{G}| \min(\ell, \log |\mathcal{G}|)^{k_1+k_2})$ time. For each batch, we use $\mathcal{O}(n + |\mathcal{G}|)$ time and space for radix sort, yielding a total of $\mathcal{O}(n + |\mathcal{G}|)$ space and $\mathcal{O}(n + |\mathcal{G}| \min(\ell, \log |\mathcal{G}|)^{k_1+k_2})$ time. Finally, for each non-empty set $\mathcal{G}'_{\delta_1,\delta_2}$ with $\mathcal{G}' \in \mathbf{G}$, each $d_1 \in [0, k_1]$, and each $d_2 \in [0, k_2]$, we spend $\mathcal{O}(|\mathcal{G}'_{\delta_1,\delta_2}|)$ time and space to generate $\mathcal{U}'$ and $\mathcal{V}'$. Since $k_1, k_2 = \mathcal{O}(1)$, the overall time of this final phase is proportional to the total size of the sets $\mathcal{G}'_{\delta_1,\delta_2}$, which is $\mathcal{O}(|\mathcal{G}| \min(\ell, \log |\mathcal{G}|)^{k_1+k_2})$ as argued above. ◀

▶ **Theorem 27.** *Consider two (ℓ, ℓ)-families $\mathcal{U}, \mathcal{V}$ of total size N consisting of pairs of substrings of a given length-n text. For any non-negative integers $k_1, k_2 = \mathcal{O}(1)$, the value*

[2] $\mathbf{G}$ is processed in batches of size $\Omega(n)$ (and not, say, set by set) so that the time required to bucket sort integers of magnitude $n^{\mathcal{O}(1)}$ in the algorithm does not make the overall time complexity worse.

$\mathrm{maxPairLCP}_{k_1,k_2}(\mathcal{U},\mathcal{V})$ *can be computed:*

- *in* $\mathcal{O}(n + N\log^{k_1+k_2+1} N)$ *time and* $\mathcal{O}(n + N)$ *space if* $\ell > \log^{3/2} N$*,*
- *in* $\mathcal{O}(n + N\ell\log^{k_1+k_2-1/2} N)$ *time and* $\mathcal{O}(n + N\ell/\log N)$ *space if* $\log N < \ell \le \log^{3/2} N$*,*
- *in* $\mathcal{O}(n + N\ell^{k_1+k_2}\sqrt{\log N})$ *time and* $\mathcal{O}(n + N)$ *space if* $\ell \le \log N$*.*

Proof. First, we augment the given text with a data structure for $\mathcal{O}(1)$-time LCP queries (see Theorem 12). Next, we apply Proposition 39 to generate a family $\mathbf{P}$ such that $\mathrm{maxPairLCP}_{k_1,k_2}(\mathcal{U},\mathcal{V}) = \max_{(\mathcal{U}',\mathcal{V}')\in\mathbf{P}} \mathrm{maxPairLCP}(\mathcal{U}',\mathcal{V}')$. We process pairs $(\mathcal{U}',\mathcal{V}') \in \mathbf{P}$ in batches $\mathbf{P}' \subseteq \mathbf{P}$ consisting of pairs of sets of total size $\Theta(N)$ (in the last batch, the total size might be smaller). Each batch $\mathbf{P}' = \{(\mathcal{U}'_j,\mathcal{V}'_j) : j \in [1,p]\}$ is processed as follows.

First, for each $j \in [1,p]$ and $i \in \{1,2\}$, we construct the compacted trie $\mathcal{T}(\mathcal{F}_{i,j})$ of the set

$$\mathcal{F}_{i,j} = \{F_i^{\Delta_i} : (F_1^{\Delta_1}, F_2^{\Delta_2}) \in \mathcal{U}'_j \cup \mathcal{V}'_j\}.$$

The two linear orders associated with the set $\mathcal{U}'_j \cup \mathcal{V}'_j$ yield the lexicographic order of the sets $\mathcal{F}_{1,j}$ and $\mathcal{F}_{2,j}$. Thus, in order to build the compacted tries $\mathcal{T}(\mathcal{F}_{i,j})$ using Lemma 6, it suffices to determine the longest common prefixes between any two consecutive modified strings in $\mathcal{F}_{i,j}$, which reduces to LCP queries on the text.

Then, for $i \in \{1,2\}$, we construct the compacted trie $\mathcal{T}(\mathcal{F}_i)$ of the set

$$\mathcal{F}_i = \bigcup_{j=1}^{p} \{j \cdot F_i^{\Delta_i} : (F_1^{\Delta_1}, F_2^{\Delta_2}) \in \mathcal{U}'_j \cup \mathcal{V}'_j\} \subseteq (\Sigma \cup [1,p])^{\le(\ell+1)}.$$

For this, we create a new root node and, for $j \in [1,m]$, attach the compacted trie $\mathcal{T}(\mathcal{F}_{i,j})$ with a length-1 edge; if the root of $\mathcal{T}(\mathcal{F}_{i,j})$ has degree 1 and $\varepsilon \notin \mathcal{F}_{i,j}$, we also dissolve the root. Finally, we construct two sets $\mathcal{P},\mathcal{Q} \subseteq \mathcal{F}_1 \times \mathcal{F}_2$:

$$\mathcal{P} = \bigcup_{j=1}^{p} \{(j \cdot U_1^{\Delta_1}, j \cdot U_2^{\Delta_2}) : (U_1^{\Delta_1}, U_2^{\Delta_2}) \in \mathcal{U}'_j\},$$

$$\mathcal{Q} = \bigcup_{j=1}^{p} \{(j \cdot V_1^{\nabla_1}, j \cdot V_2^{\nabla_2}) : (V_1^{\nabla_1}, V_2^{\nabla_2}) \in \mathcal{V}'_j\}.$$

This yields an instance of the Two String Families LCP Problem, which we solve using Lemma 3 if $\ell > \log^{3/2} N$ or Lemma 4 otherwise. The obtained value satisfies $\mathrm{maxPairLCP}(\mathcal{P},\mathcal{Q}) = 2 + \max_{j=1}^{m} \mathrm{maxPairLCP}(\mathcal{U}'_j,\mathcal{V}'_j)$ and, taking the maximum across all batches $\mathbf{P}' \subseteq \mathbf{P}$, we retrieve $\mathrm{maxPairLCP}_{k_1,k_2}(\mathcal{U},\mathcal{V}) = \max_{(\mathcal{U}',\mathcal{V}')\in\mathbf{P}} \mathrm{maxPairLCP}(\mathcal{U}',\mathcal{V}')$.

We conclude with the complexity analysis. Applying the algorithm of Proposition 39 costs $\mathcal{O}(n + N\min(\ell,\log N)^{k_1+k_2})$ time and $\mathcal{O}(n + N)$ space. The resulting family $\mathbf{P}$ satisfies $\max_{(\mathcal{U}',\mathcal{V}')\in\mathbf{P}}(|\mathcal{U}'|+|\mathcal{V}'|) = \mathcal{O}(N)$ and $\sum_{(\mathcal{U}',\mathcal{V}')\in\mathbf{P}} = \mathcal{O}(N\min(\ell,\log N)^{k_1+k_2})$. Hence, the number of batches $\mathbf{P}'$ is $\mathcal{O}(\min(\ell,\log N)^{k_1+k_2})$. Let us now focus on a single batch $\mathbf{P}' = \{(\mathcal{U}'_j,\mathcal{V}'_j) : 1 \le j \le p\}$; recall that it satisfies $\sum_{j=1}^{p}(|\mathcal{U}'_j| + |\mathcal{V}'_j|) = \mathcal{O}(N)$. Each set $\mathcal{F}_{i,j}$ (with strings sorted lexicographically) is obtained from $\mathcal{U}'_j \cup \mathcal{V}'_j$ in $\mathcal{O}(|\mathcal{U}'_j| + |\mathcal{V}'_j|)$ time. Each modified string in $\mathcal{F}_{i,j}$ contains up to k_i modifications, so the LCP computation for two such modified strings requires up to $2k_i + 1 = \mathcal{O}(1)$ LCP queries on T. Consequently, each trie $\mathcal{T}(\mathcal{F}_{i,j})$ is constructed in $\mathcal{O}(|\mathcal{U}'_j| + |\mathcal{V}'_j|)$ time. Merging these tries into $\mathcal{T}(\mathcal{F}_i)$ requires $\mathcal{O}(p)$ additional time, i.e., the construction of $\mathcal{T}(\mathcal{F}_i)$ takes $\mathcal{O}(N)$ time in total. The sets $\mathcal{P}$ and $\mathcal{Q}$ are of size $\mathcal{O}(N)$ and also take $\mathcal{O}(N)$ time to construct. Overall, preparing the instance $(\mathcal{T}(\mathcal{F}_1),\mathcal{T}(\mathcal{F}_2),\mathcal{P},\mathcal{Q})$ of Two String Families LCP Problem takes $\mathcal{O}(N)$ time and space. Solving this instance takes $\mathcal{O}(N\log N)$ time and $\mathcal{O}(N)$ space if we use Lemma 3 or $\mathcal{O}(N(\ell+\log N)(\log\ell+\sqrt{\log N})/\log N)$ time and $\mathcal{O}(N+N\ell/\log N)$ space if we use Lemma 4.

In either case, this final step dominates both the time and the space complexity of processing a single batch $\mathbf{P}'$. Across all $\mathcal{O}(\min(\ell, \log N)^{k_1+k_2})$ batches, we obtain the following trade-offs:

- $\mathcal{O}(N \log N \cdot \log^{k_1+k_2} N)$ time and $\mathcal{O}(N)$ space if $\ell > \log^{3/2} N$;
- $\mathcal{O}(N\ell/\sqrt{\log N} \cdot \log^{k_1+k_2} N)$ time and $\mathcal{O}(N\ell/\log N)$ space if $\log N < \ell \leq \log^{3/2} N$;
- $\mathcal{O}(N\sqrt{\log N} \cdot \ell^{k_1+k_2})$ time and $\mathcal{O}(N)$ space if $\ell \leq \log N$.

Accounting for $\mathcal{O}(n)$ space and construction time of the data structure for LCP queries on the text, we retrieve the claimed trade-offs. ◀

References

1 Amir Abboud, Richard Ryan Williams, and Huacheng Yu. More applications of the polynomial method to algorithm design. In Piotr Indyk, editor, *26th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015*, pages 218–230. SIAM, 2015. doi:10.1137/1.9781611973730.17.

2 Amihood Amir and Itai Boneh. Locally maximal common factors as a tool for efficient dynamic string algorithms. In Gonzalo Navarro, David Sankoff, and Binhai Zhu, editors, *Annual Symposium on Combinatorial Pattern Matching, CPM 2018, July 2-4, 2018 - Qingdao, China*, volume 105 of *LIPIcs*, pages 11:1–11:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPIcs.CPM.2018.11.

3 Amihood Amir, Panagiotis Charalampopoulos, Solon P. Pissis, and Jakub Radoszewski. Dynamic and internal longest common substring. *Algorithmica*, 82(12):3707–3743, 2020. doi:10.1007/s00453-020-00744-0.

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

5 Maxim A. Babenko, Paweł Gawrychowski, Tomasz Kociumaka, and Tatiana Starikovskaya. Wavelet trees meet suffix trees. In Piotr Indyk, editor, *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015, San Diego, CA, USA, January 4-6, 2015*, pages 572–591. SIAM, 2015. doi:10.1137/1.9781611973730.39.

6 Maxim A. Babenko and Tatiana Starikovskaya. Computing the longest common substring with one mismatch. *Probl. Inf. Transm.*, 47(1):28–33, 2011. doi:10.1134/S0032946011010030.

7 Ricardo A. Baeza-Yates. Improved string searching. *Softw. Pract. Exp.*, 19(3):257–271, 1989. doi:10.1002/spe.4380190305.

8 Hideo Bannai, Tomohiro I, Shunsuke Inenaga, Yuto Nakashima, Masayuki Takeda, and Kazuya Tsuruta. The "runs" theorem. *SIAM J. Comput.*, 46(5):1501–1514, 2017. doi:10.1137/15M1011032.

9 Djamal Belazzougui. Worst case efficient single and multiple string matching in the RAM model. In Costas S. Iliopoulos and William F. Smyth, editors, *Combinatorial Algorithms - 21st International Workshop, IWOCA 2010, London, UK, July 26-28, 2010, Revised Selected Papers*, volume 6460 of *Lecture Notes in Computer Science*, pages 90–102. Springer, 2010. doi:10.1007/978-3-642-19222-7_10.

10 Djamal Belazzougui. Improved space-time tradeoffs for approximate full-text indexing with one edit error. *Algorithmica*, 72(3):791–817, 2015. doi:10.1007/s00453-014-9873-9.

11 Oren Ben-Kiki, Philip Bille, Dany Breslauer, Leszek Gasieniec, Roberto Grossi, and Oren Weimann. Optimal packed string matching. In Supratik Chakraborty and Amit Kumar, editors, *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2011, December 12-14, 2011, Mumbai, India*, volume 13 of *LIPIcs*, pages 423–432. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2011. doi:10.4230/LIPIcs.FSTTCS.2011.423.

12 Oren Ben-Kiki, Philip Bille, Dany Breslauer, Leszek Gasieniec, Roberto Grossi, and Oren Weimann. Towards optimal packed string matching. *Theor. Comput. Sci.*, 525:111–129, 2014. doi:10.1016/j.tcs.2013.06.013.

13 Stav Ben-Nun, Shay Golan, Tomasz Kociumaka, and Matan Kraus. Time-space tradeoffs for finding a long common substring. In Inge Li Gørtz and Oren Weimann, editors, *31st Annual Symposium on Combinatorial Pattern Matching, CPM 2020, June 17-19, 2020, Copenhagen, Denmark*, volume 161 of *LIPIcs*, pages 5:1–5:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.CPM.2020.5.

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

15 Philip Bille. Fast searching in packed strings. In Gregory Kucherov and Esko Ukkonen, editors, *Combinatorial Pattern Matching, 20th Annual Symposium, CPM 2009, Lille, France, June 22-24, 2009, Proceedings*, volume 5577 of *Lecture Notes in Computer Science*, pages 116–126. Springer, 2009. doi:10.1007/978-3-642-02441-2_11.

16 Philip Bille. Fast searching in packed strings. *J. Discrete Algorithms*, 9(1):49–56, 2011. doi:10.1016/j.jda.2010.09.003.

17 Philip Bille, Inge Li Gørtz, and Frederik Rye Skjoldjensen. Deterministic indexing for packed strings. In Juha Kärkkäinen, Jakub Radoszewski, and Wojciech Rytter, editors, *28th Annual Symposium on Combinatorial Pattern Matching, CPM 2017, July 4-6, 2017, Warsaw, Poland*, volume 78 of *LIPIcs*, pages 6:1–6:11. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.CPM.2017.6.

18 Dany Breslauer, Leszek Gasieniec, and Roberto Grossi. Constant-time word-size string matching. In Juha Kärkkäinen and Jens Stoye, editors, *Combinatorial Pattern Matching - 23rd Annual Symposium, CPM 2012, Helsinki, Finland, July 3-5, 2012. Proceedings*, volume 7354 of *Lecture Notes in Computer Science*, pages 83–96. Springer, 2012. doi:10.1007/978-3-642-31265-6_7.

19 Karl Bringmann, Philip Wellnitz, and Marvin Künnemann. Few matches or almost periodicity: Faster pattern matching with mismatches in compressed texts. In Timothy M. Chan, editor, *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019, San Diego, California, USA, January 6-9, 2019*, pages 1126–1145. SIAM, 2019. doi:10.1137/1.9781611975482.69.

20 Gerth Stølting Brodal and Christian N. S. Pedersen. Finding maximal quasiperiodicities in strings. In Raffaele Giancarlo and David Sankoff, editors, *Combinatorial Pattern Matching, 11th Annual Symposium, CPM 2000, Montreal, Canada, June 21-23, 2000, Proceedings*, volume 1848 of *Lecture Notes in Computer Science*, pages 397–411. Springer, 2000. doi:10.1007/3-540-45123-4_33.

21 Mark R. Brown and Robert Endre Tarjan. A fast merging algorithm. *J. ACM*, 26(2):211–226, 1979. doi:10.1145/322123.322127.

22 Stefan Burkhardt and Juha Kärkkäinen. Fast lightweight suffix array construction and checking. In Ricardo A. Baeza-Yates, Edgar Chávez, and Maxime Crochemore, editors, *Combinatorial Pattern Matching, CPM 2003*, volume 2676 of *LNCS*, pages 55–69. Springer, 2003. doi:10.1007/3-540-44888-8_5.

23 Domenico Cantone and Simone Faro. Pattern matching with swaps for short patterns in linear time. In Mogens Nielsen, Antonín Kucera, Peter Bro Miltersen, Catuscia Palamidessi, Petr Tuma, and Frank D. Valencia, editors, *SOFSEM 2009: Theory and Practice of Computer Science, 35th Conference on Current Trends in Theory and Practice of Computer Science, Spindleruv Mlýn, Czech Republic, January 24-30, 2009. Proceedings*, volume 5404 of *Lecture Notes in Computer Science*, pages 255–266. Springer, 2009. doi:10.1007/978-3-540-95891-8_25.

24 Ho-Leung Chan, Tak Wah Lam, Wing-Kin Sung, Siu-Lung Tam, and Swee-Seong Wong. Compressed indexes for approximate string matching. *Algorithmica*, 58(2):263–281, 2010. doi:10.1007/s00453-008-9263-2.

25 Ho-Leung Chan, Tak Wah Lam, Wing-Kin Sung, Siu-Lung Tam, and Swee-Seong Wong. A linear size index for approximate pattern matching. *J. Discrete Algorithms*, 9(4):358–364, 2011. doi:10.1016/j.jda.2011.04.004.

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

27 Panagiotis Charalampopoulos, Paweł Gawrychowski, and Karol Pokorski. Dynamic longest common substring in polylogarithmic time. In Artur Czumaj, Anuj Dawar, and Emanuela Merelli, editors, *47th International Colloquium on Automata, Languages, and Programming, ICALP 2020, July 8-11, 2020, Saarbrücken, Germany (Virtual Conference)*, volume 168 of *LIPIcs*, pages 27:1–27:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.ICALP.2020.27.

28 Panagiotis Charalampopoulos, Costas S. Iliopoulos, Chang Liu, and Solon P. Pissis. Property suffix array with applications in indexing weighted sequences. *ACM J. Exp. Algorithmics*, 25:1–16, 2020. doi:10.1145/3385898.

29 Panagiotis Charalampopoulos, Tomasz Kociumaka, Solon P. Pissis, Jakub Radoszewski, Wojciech Rytter, Juliusz Straszyński, Tomasz Waleń, and Wiktor Zuba. Circular pattern matching with k mismatches. *J. Comput. Syst. Sci.*, 115:73–85, 2021. doi:10.1016/j.jcss.2020.07.003.

30 Archie L. Cobbs. Fast approximate matching using suffix trees. In Zvi Galil and Esko Ukkonen, editors, *Combinatorial Pattern Matching, 6th Annual Symposium, CPM 95, Espoo, Finland, July 5-7, 1995, Proceedings*, volume 937 of *Lecture Notes in Computer Science*, pages 41–54. Springer, 1995. doi:10.1007/3-540-60044-2_33.

31 Vincent Cohen-Addad, Laurent Feuilloley, and Tatiana Starikovskaya. Lower bounds for text indexing with mismatches and differences. In Timothy M. Chan, editor, *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019, San Diego, California, USA, January 6-9, 2019*, pages 1146–1164. SIAM, 2019. doi:10.1137/1.9781611975482.70.

32 Richard Cole, Lee-Ad Gottlieb, and Moshe Lewenstein. Dictionary matching and indexing with errors and don't cares. In László Babai, editor, *Proceedings of the 36th Annual ACM Symposium on Theory of Computing, Chicago, IL, USA, June 13-16, 2004*, pages 91–100. ACM, 2004. doi:10.1145/1007352.1007374.

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

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

35 Martin Farach. Optimal suffix tree construction with large alphabets. In *38th Annual Symposium on Foundations of Computer Science, FOCS '97, Miami Beach, Florida, USA, October 19-22, 1997*, pages 137–143. IEEE Computer Society, 1997. doi:10.1109/SFCS.1997.646102.

36 Nathan J. Fine and Herbert S. Wilf. Uniqueness theorems for periodic functions. *Proceedings of the American Mathematical Society*, 16(1):109–114, 1965. URL: http://www.jstor.org/stable/2034009.

37 Tomás Flouri, Emanuele Giaquinta, Kassian Kobert, and Esko Ukkonen. Longest common substrings with k mismatches. *Inf. Process. Lett.*, 115(6-8):643–647, 2015. doi:10.1016/j.ipl.2015.03.006.

38 Kimmo Fredriksson. Faster string matching with super-alphabets. In Alberto H. F. Laender and Arlindo L. Oliveira, editors, *String Processing and Information Retrieval, 9th International Symposium, SPIRE 2002, Lisbon, Portugal, September 11-13, 2002, Proceedings*, volume 2476 of *Lecture Notes in Computer Science*, pages 44–57. Springer, 2002. doi:10.1007/3-540-45735-6_5.

39 Kimmo Fredriksson. Shift-or string matching with super-alphabets. *Inf. Process. Lett.*, 87(4):201–204, 2003. doi:10.1016/S0020-0190(03)00296-5.

40 Harold N. Gabow and Robert Endre Tarjan. A linear-time algorithm for a special case of disjoint set union. *J. Comput. Syst. Sci.*, 30(2):209–221, 1985. doi:10.1016/0022-0000(85)90014-5.

41 François Le Gall and Saeed Seddighin. Quantum meets fine-grained complexity: Sublinear time quantum algorithms for string problems. *CoRR*, abs/2010.12122, 2020. arXiv:2010.12122.

42 Emanuele Giaquinta, Szymon Grabowski, and Kimmo Fredriksson. Approximate pattern matching with k-mismatches in packed text. *Inf. Process. Lett.*, 113(19-21):693–697, 2013. doi:10.1016/j.ipl.2013.07.002.

43 Szymon Grabowski. A note on the longest common substring with k-mismatches problem. *Inf. Process. Lett.*, 115(6-8):640–642, 2015. doi:10.1016/j.ipl.2015.03.003.

44 Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms, January 12-14, 2003, Baltimore, Maryland, USA*, pages 841–850. ACM/SIAM, 2003. URL: http://dl.acm.org/citation.cfm?id=644108.644250.

45 Yijie Han. Deterministic sorting in $O(\log\log n)$ time and linear space. *J. Algorithms*, 50(1):96–105, 2004. doi:10.1016/j.jalgor.2003.09.001.

46 Lucas Chi Kwong Hui. Color set size problem with application to string matching. In Alberto Apostolico, Maxime Crochemore, Zvi Galil, and Udi Manber, editors, *Combinatorial Pattern Matching, Third Annual Symposium, CPM 92, Tucson, Arizona, USA, April 29 - May 1, 1992, Proceedings*, volume 644 of *Lecture Notes in Computer Science*, pages 230–243. Springer, 1992. doi:10.1007/3-540-56024-6_19.

47 Trinh N. D. Huynh, Wing-Kai Hon, Tak Wah Lam, and Wing-Kin Sung. Approximate string matching using compressed suffix arrays. *Theor. Comput. Sci.*, 352(1-3):240–249, 2006. doi:10.1016/j.tcs.2005.11.022.

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

49 Russell Impagliazzo, Ramamohan Paturi, and Francis Zane. Which problems have strongly exponential complexity? *Journal of Computer and System Sciences*, 63(4):512–530, 2001. doi:10.1006/jcss.2001.1774.

50 Dominik Kempa and Tomasz Kociumaka. String synchronizing sets: sublinear-time BWT construction and optimal LCE data structure. In *51st Annual ACM SIGACT Symposium on Theory of Computing, STOC 2019*, pages 756–767. ACM, 2019. doi:10.1145/3313276.3316368.

51 Shmuel Tomi Klein and Miri Ben-Nissan. Accelerating Boyer Moore searches on binary texts. In Jan Holub and Jan Zdárek, editors, *Implementation and Application of Automata, 12th International Conference, CIAA 2007, Prague, Czech Republic, July 16-18, 2007, Revised Selected Papers*, volume 4783 of *Lecture Notes in Computer Science*, pages 130–143. Springer, 2007. doi:10.1007/978-3-540-76336-9_14.

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

53 Tomasz Kociumaka, Tatiana Starikovskaya, and Hjalte Wedel Vildhøj. Sublinear space algorithms for the longest common substring problem. In *Algorithms - ESA 2014 - 22th*

Annual European Symposium, Wroclaw, Poland, September 8-10, 2014. Proceedings, pages 605–617, 2014. doi:10.1007/978-3-662-44777-2_50.

54 Roman M. Kolpakov and Gregory Kucherov. Finding maximal repetitions in a word in linear time. In *40th Annual Symposium on Foundations of Computer Science, FOCS '99, 17-18 October, 1999, New York, NY, USA*, pages 596–604. IEEE Computer Society, 1999. doi:10.1109/SFFCS.1999.814634.

55 Gad M. Landau and Uzi Vishkin. Fast string matching with k differences. *J. Comput. Syst. Sci.*, 37(1):63–78, 1988. doi:10.1016/0022-0000(88)90045-1.

56 Mamoru Maekawa. A $\sqrt{n}$ algorithm for mutual exclusion in decentralized systems. *ACM Transactions on Computer Systems*, 3(2):145–159, 1985. doi:10.1145/214438.214445.

57 J. Ian Munro, Gonzalo Navarro, and Yakov Nekrich. Text indexing and searching in sublinear time. In Inge Li Gørtz and Oren Weimann, editors, *31st Annual Symposium on Combinatorial Pattern Matching, CPM 2020, June 17-19, 2020, Copenhagen, Denmark*, volume 161 of *LIPIcs*, pages 24:1–24:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.CPM.2020.24.

58 J. Ian Munro, Yakov Nekrich, and Jeffrey Scott Vitter. Fast construction of wavelet trees. *Theor. Comput. Sci.*, 638:91–97, 2016. doi:10.1016/j.tcs.2015.11.011.

59 Gonzalo Navarro and Mathieu Raffinot. A bit-parallel approach to suffix automata: Fast extended string matching. In Martin Farach-Colton, editor, *Combinatorial Pattern Matching, 9th Annual Symposium, CPM 98, Piscataway, New Jersey, USA, July 20-22, 1998, Proceedings*, volume 1448 of *Lecture Notes in Computer Science*, pages 14–33. Springer, 1998. doi:10.1007/BFb0030778.

60 Tatiana Starikovskaya and Hjalte Wedel Vildhøj. Time-space trade-offs for the longest common substring problem. In Johannes Fischer and Peter Sanders, editors, *Combinatorial Pattern Matching, 24th Annual Symposium, CPM 2013, Bad Herrenalb, Germany, June 17-19, 2013. Proceedings*, volume 7922 of *Lecture Notes in Computer Science*, pages 223–234. Springer, 2013. doi:10.1007/978-3-642-38905-4_22.

61 Jorma Tarhio and Hannu Peltola. String matching in the DNA alphabet. *Softw. Pract. Exp.*, 27(7):851–861, 1997.

62 Sharma V. Thankachan, Chaitanya Aluru, Sriram P. Chockalingam, and Srinivas Aluru. Algorithmic framework for approximate matching under bounded edits with applications to sequence analysis. In Benjamin J. Raphael, editor, *Research in Computational Molecular Biology - 22nd Annual International Conference, RECOMB 2018, Paris, France, April 21-24, 2018, Proceedings*, volume 10812 of *Lecture Notes in Computer Science*, pages 211–224. Springer, 2018. doi:10.1007/978-3-319-89929-9_14.

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

64 Dekel Tsur. Fast index for approximate string matching. *J. Discrete Algorithms*, 8(4):339–345, 2010. doi:10.1016/j.jda.2010.08.002.

65 Esko Ukkonen. Approximate string-matching over suffix trees. In Alberto Apostolico, Maxime Crochemore, Zvi Galil, and Udi Manber, editors, *Combinatorial Pattern Matching, 4th Annual Symposium, CPM 93, Padova, Italy, June 2-4, 1993, Proceedings*, volume 684 of *Lecture Notes in Computer Science*, pages 228–242. Springer, 1993. doi:10.1007/BFb0029808.

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

A Proof of Proposition 35

Our construction follows [63, Section 4] along with the efficient implementation described in [63, Section 5.1]. However, we cannot use these results in a black-box manner since we are solving a slightly more general problem: In [63], the input consists of two strings X and Y, and the output family $\mathbf{F}$ must satisfy the following condition: for every suffix U of X and every suffix V of Y, there exists a set $\mathcal{F}' \in \mathbf{F}$ and modified strings $U^{\Delta}, V^{\nabla} \in \mathcal{F}'$ forming a $(U,V)_k$-maxpair. Instead, in Proposition 35, we have a set $\mathcal{F}$ of (selected) substrings of a single text, and the output family $\mathbf{F}$ must be a k-complete family for $\mathcal{F}$, i.e., satisfy the following condition: for every $U, V \in \mathcal{F}$, there exists a set $\mathcal{F}' \in \mathbf{F}$ and modified strings $U^{\Delta}, V^{\nabla} \in \mathcal{F}'$ forming a $(U,V)_k$-maxpair. Additionally, we need stronger bounds on the sizes of sets $\mathcal{F}' \in \mathbf{F}$. The original argument [63, Lemma 3] yields $\max_{\mathcal{F}'\in\mathbf{F}} |\mathcal{F}'| = \mathcal{O}(|\mathcal{F}|)$ and $\sum_{\mathcal{F}'\in\mathbf{F}} |\mathcal{F}'| = \mathcal{O}(|\mathcal{F}| \log^k |\mathcal{F}|)$. Instead, we need to prove that each string $F \in \mathcal{F}$ is the source of $\mathcal{O}(1)$ modified strings in any single set $\mathcal{F}' \in \mathbf{F}$ and that there are $\mathcal{O}(\min(\ell, \log |\mathcal{F}|)^k)$ modified strings across all sets $\mathcal{F}' \in \mathbf{F}$ provided that $\mathcal{F} \subseteq \Sigma^{\le \ell}$.

On the high-level, the algorithm behind Proposition 35 constructs d-complete families $\mathbf{F}_d$ for $\mathcal{F}$ for each $d \in [0,k]$. In this setting, the input set $\mathcal{F}$ is interpreted as a 0-complete family $\mathbf{F}_0 = \{\mathcal{F}\}$. For the sake of space efficiency, all families are built in parallel, and the construction algorithm is organised into $2k+1$ *levels*, indexed with 0 through $2k$. Each even level $2d$ is given a stream of sets $\mathcal{F}' \in \mathbf{F}_d$ and its task is to construct the compacted trie $\mathcal{T}(\mathcal{F}')$ for each given $\mathcal{F}' \in \mathbf{F}_d$ (which, in particular, involves sorting the modified strings in $\mathcal{F}'$). Each odd level $2d-1$ is given a stream of tries $\mathcal{T}(\mathcal{F}')$ for $\mathcal{F}' \in \mathbf{F}_{d-1}$, and its task is to output a stream of sets $\mathcal{F}'' \in \mathbf{F}_d$.

The claimed properties of the constructed k-complete family $\mathbf{F}_k$ naturally generalise to analogous properties of the intermediate d-complete families $\mathbf{F}_d$. Moreover, efficient implementation of even levels relies on the following additional invariant:

▶ **Invariant 41.** *Every set $\mathcal{F}' \in \mathbf{F}_d$ contains a* pivot *$P^{\nabla} \in \mathcal{F}'$ such that, for every $F^{\Delta} \in \mathcal{F}'$, we have $\Delta \subseteq [1, \mathsf{LCP}(F^{\Delta}, P^{\nabla})] \times \Sigma$.*

The invariant is trivially true for $d = 0$ due to $\Delta = \emptyset$ for every $F^{\Delta} \in \mathcal{F}'$.

A.1 Implementation and Analysis of Even Levels

Recall that the goal of an even level $2d$ is to construct $\mathcal{T}(\mathcal{F}')$ for each $\mathcal{F}' \in \mathbf{F}_d$. For this, we rely on Lemma 6, which requires sorting the modified strings in $F^{\Delta} \in \mathcal{F}'$ and computing the longest common prefixes between pairs of consecutive strings.

For the latter task, in the preprocessing, we augment the input text with a data structure for $\mathcal{O}(1)$-time LCP queries (Theorem 12). Since each modified string $F^{\Delta} \in \mathcal{F}'$ satisfies $|\Delta| \le d$, the longest common prefix of any two modified strings in $\mathcal{F}'$ can be computed in $\mathcal{O}(1)$ time using up to $2d+1$ LCP queries. In particular, this yields $\mathcal{O}(1)$-time lexicographic comparison of any two modified strings in $\mathcal{F}'$, which means that $\mathcal{F}'$ can be sorted in $\mathcal{O}(|\mathcal{F}'| \log |\mathcal{F}'|)$ time. Nevertheless, this is too much for our purposes. A more efficient procedure requires buffering the sets $\mathcal{F}' \in \mathbf{F}_d$ into batches $\mathbf{F}'_d \subseteq \mathbf{F}_d$ of total size $\Theta(n + |\mathcal{F}|)$ and using the following result:

▶ **Theorem 42** ([28, Theorem 12])**.** *Any m substrings of a given length-n text can be sorted lexicographically in $\mathcal{O}(n+m)$ time.*

Unfortunately, the possibility of generalising this result to modified substrings remains an open question. A workaround proposed in [63] relies on Invariant 41. A pivot $P^{\nabla} \in \mathcal{F}'$ can be retrieved in $\mathcal{O}(|\mathcal{F}'|)$ time as any modified string $F^{\Delta} \in \mathcal{F}'$ for which $\max\{i : (i,c) \in \Delta\}$ is maximised. In the lexicographic order of $\mathcal{F}'$, the strings satisfying $F^\Delta \le P^\nabla$ come before those satisfying $F^\Delta > P^\nabla$. Moreover, in the former group, the values $\mathsf{LCP}(F^\Delta, P^\nabla)$ form a non-decreasing sequence and, in the latter group, these values form a non-increasing sequence. This way, the task of sorting $\mathcal{F}'$ reduces to partitioning $\mathcal{F}'$ into $\mathcal{F}'_\ell = \{F^\Delta \in \mathcal{F}' : \mathsf{LCP}(F^\Delta, P^\nabla) = \ell\}$ and sorting each set $\mathcal{F}'_\ell$. We first compute $\mathsf{LCP}(F^\Delta, P^\nabla)$ for all $F^\Delta \in \mathcal{F}'$ in $\mathcal{O}(|\mathcal{F}'|)$ time. Then, in order to construct sets $\mathcal{F}'_\ell$, we use radix sort for the whole batch $\mathbf{F}'_d \subseteq \mathbf{F}_d$—the keys ($\mathsf{LCP}(F^\Delta, P^\nabla)$ for $F^\Delta \in \mathcal{F}'$) are integers bounded by n. In order to sort the elements of each set $\mathcal{F}'_\ell$, we exploit the fact that the lexicographic order of a modified string $F^\Delta \in \mathcal{F}'_\ell$ is determined by $F^\Delta[\ell+1\,..\,|F|]$. Moreover, as $\ell = \mathsf{LCP}(F^\Delta, P^\nabla)$ for the pivot P^∇, we have $\Delta \subseteq [1,\ell] \times \Sigma$ by Invariant 41 and thus $F^\Delta[\ell+1\,..\,|F|] = F[\ell+1\,..\,|F|]$. Thus, $F^\Delta[\ell+1\,..\,|F|]$ is a substring of the input text and hence we can use Theorem 42 to sort all such substrings arising as we process the batch $\mathbf{F}'_d \subseteq \mathbf{F}_d$, and then classify them back into individual subsets $\mathcal{F}'_\ell$ using radix sort.

The overall time and space complexity for processing a single batch $\mathbf{F}'_d \subseteq \mathbf{F}_d$ is therefore $\mathcal{O}(n + |\mathcal{F}|)$. Across all batches, the space remains $\mathcal{O}(n + |\mathcal{F}|)$ and the running time becomes $\mathcal{O}(n + |\mathcal{F}| \min(\ell, \log |\mathcal{F}|)^d)$ due to $\sum_{\mathcal{F}' \in \mathbf{F}_d} |\mathcal{F}'| = \mathcal{O}(|\mathcal{F}| \min(\ell, \log|\mathcal{F}|)^d)$.

A.2 Implementation and Analysis of Odd Levels

Recall that the goal of an odd level $2d+1$ is to transform a stream of tries $\mathcal{T}(\mathcal{F}')$ for $\mathcal{F}' \in \mathbf{F}_{d-1}$ into a stream of sets $\mathcal{F}'' \in \mathbf{F}_d$. This process is guided by the *heavy-light decomposition* of $\mathcal{T}(\mathcal{F}')$, which classifies the nodes of $\mathcal{T}(\mathcal{F}')$ into *heavy* and *light* so that the root is light and exactly one child of each internal node is heavy: the one whose subtree contains the maximum number of leaves (with ties broken arbitrarily). Each light node w is therefore associated with the *heavy path* which starts at w and repeatedly proceeds to the unique heavy child until reaching a leaf, which we denote $h(w)$. The key property of the heavy-light decomposition is that each node has $\mathcal{O}(\log|\mathcal{F}'|)$ light ancestors. However, since the height of $\mathcal{T}(\mathcal{F}')$ is $\mathcal{O}(\ell)$, we can also bound the number of light ancestors by $\mathcal{O}(\min(\ell, \log|\mathcal{F}'|))$.

The algorithm constructs the heavy-light decomposition of $\mathcal{T}(\mathcal{F}')$ and, for each light node w, creates a set $\mathcal{F}'_w \in \mathbf{F}_d$ constructed as follows. We traverse the subtree of w and, for each modified string $F^\Delta \in \mathcal{F}'$ prefixed by $\mathsf{val}(w)$, we compute $\ell = \mathsf{LCP}(F^\Delta, \mathsf{val}(h(w)))$, which is the string depth of the lowest common ancestor of $h(w)$ and the locus of F^Δ. If $\Delta \subseteq [1,\ell] \times \Sigma$, we add F^Δ to $\mathcal{F}'_w$. If additionally $|F^\Delta| > \ell$, we also add $F^{\Delta \cup \{(\ell+1, \mathsf{val}(h(w))[\ell+1])\}}$ to $\mathcal{F}'_w$. This way, we guarantee that $\mathsf{val}(h(w)) \in \mathcal{F}'_w$ forms a pivot of $\mathcal{F}'_w$, as defined in Invariant 41. Note that each string $F^\Delta \in \mathcal{F}'$ is prefixed by $\mathsf{val}(w)$ for $\mathcal{O}(\min(\ell, \log|\mathcal{F}'|))$ light nodes w. Hence, each trie $\mathcal{T}(\mathcal{F}')$ for $\mathcal{F}' \in \mathbf{F}_{d-1}$ is processed in $\mathcal{O}(|\mathcal{F}'| \min(\ell, \log|\mathcal{F}'|)) = \mathcal{O}(|\mathcal{F}'| \min(\ell, \log|\mathcal{F}|))$ time and $\mathcal{O}(|\mathcal{F}'|) = \mathcal{O}(|\mathcal{F}|)$ space. Across all $\mathcal{F}' \in \mathbf{F}_{d-1}$, this yields $\mathcal{O}(|\mathcal{F}| \min(\ell, \log|\mathcal{F}|)^d)$ time and $\mathcal{O}(|\mathcal{F}|)$ space.

It remains to prove that $\mathbf{F}_d$ satisfies all the conditions of Proposition 35. Each modified string $F^\Delta \in \mathcal{F}'$ gives rise to at most two modified strings added to a single set $\mathcal{F}'_w$ and $\mathcal{O}(\min(\ell, \log|\mathcal{F}|))$ modified strings across sets $\mathcal{F}'_w$ for light nodes w in $\mathcal{T}(\mathcal{F}')$. Since each string $F \in \mathcal{F}$ is the source of $\mathcal{O}(1)$ modified strings in any single set $\mathcal{F}' \in \mathbf{F}_{d-1}$ and $\mathcal{O}(\min(\ell, \log|\mathcal{F}|)^{d-1})$ modified strings across all sets $\mathcal{F}' \in \mathbf{F}_{d-1}$, it is the source of $\mathcal{O}(1)$ modified strings in any single set $\mathcal{F}'_w \in \mathbf{F}_d$ and $\mathcal{O}(\min(\ell, \log|\mathcal{F}|)^d)$ modified strings across all sets $\mathcal{F}'_w \in \mathbf{F}_d$.

Finally, we shall argue that $\mathbf{F}_d$ is indeed a d-complete family. The modified strings in $\mathcal{F}' \in \mathbf{F}_{d-1}$ have sources in $\mathcal{F}$ and up to $d-1$ modifications. Each string inserted to $\mathcal{F}'_w \in \mathbf{F}_d$ retains its source and may have up to one more modification, for a total of up to d. Next, consider two strings $U, V \in \mathcal{F}$. Since $\mathbf{F}_{d-1}$ is a $(d-1)$-complete family, there is a set $\mathcal{F}' \in \mathbf{F}_{d-1}$ and modified strings $U^{\Delta}, V^{\nabla} \in \mathcal{F}'$ forming a $(U,V)_{d-1}$-maxpair. Let v be the node of $\mathcal{T}(\mathcal{F}')$ representing the longest common prefix of U^{Δ} and V^{∇}, and let w be the lowest light ancestor of v (so that v lies on the path from w to $h(w)$). By Fact 32, we have $p := \mathsf{LCP}(U^{\Delta}, V^{\nabla}) = \mathsf{LCP}_{d-1}(U,V)$. Definition 30 further yields $\Delta, \nabla \subseteq [1,p] \times \Sigma$. If $\mathsf{LCP}_d(U,V) = p$, then $U^{\Delta}, V^{\nabla} \in \mathcal{F}'_w$ form a $(U,V)_d$-maxpair. Otherwise, let $c = \mathsf{val}(h(w))[p+1]$ and note that $U[p+1] \neq V[p+1]$. If $U[p+1] \neq c \neq V[p+1]$, then $U^{\Delta\cup\{(p+1,c)\}}, V^{\nabla\cup\{(p+1,c)\}} \in \mathcal{F}'_w$ form a $(U,V)_d$-maxpair. If $U[p+1] \neq c = V[p+1]$, then $U^{\Delta\cup\{(p+1,c)\}}, V^{\nabla} \in \mathcal{F}'_w$ form a $(U,V)_d$-maxpair. Symmetrically, if $U[p+1] = c \neq V[p+1]$, then $U^{\Delta}, V^{\nabla\cup\{(p+1,c)\}} \in \mathcal{F}'_w$ form a $(U,V)_d$-maxpair. Thus, in all four cases, $\mathcal{F}'_w \in \mathbf{F}_d$ contains a $(U,V)_d$-maxpair. This completes the proof of Proposition 35.