Exploring the Gap Between LCS and LCStr

Shay Golan
Ariel University, Israel

Matan Kraus
Bar-Ilan University, Ramat Gan, Israel

Ely Porat
Bar-Ilan University, Ramat Gan, Israel

B. Riva Shalom
Shenkar College, Ramat Gan, Israel

Abstract

The Longest Common Subsequence (LCS) problem and the Longest Common Substring (LCStr) problem are classical string problems with broad theoretical and practical significance. The former has a quadratic conditional lower bound [FOCS, 2015], while the latter admits a linear-time solution.

In this paper, we study a natural variation of these problems, the *Longest Common Subsequence-Substring (LCSS)* problem. The LCSS problem seeks the longest string that is simultaneously a subsequence of one input string and a substring of the other. This variant bridges LCS and LCStr, raising intriguing algorithmic questions: Does the complexity of computing LCSS interpolate between the linear time of LCStr and the quadratic time of LCS? What about approximability? We also examine a natural extension of LCSS to multiple strings, parameterizing the balance between subsequence and substring requirements.

Our results reveal several insights. First, under the SETH conjecture, the inherent complexity of LCSS is quadratic, similar to LCS. In contrast, we provide a linear-time approximation for LCSS. Finally, for the multi-string variant, unlike both problems, we design a quadratic-time algorithm, uncovering deeper structural properties of the problem.

By studying the complexity of the LCSS problem, we aim to gain some understanding of what influences whether a variant of the LCS problem behaves more like the standard LCS or like LCStr. Our findings suggest that hybrid constraints can create computational "sweet spots," where problems become more tractable than their pure counterparts. This opens a broader research direction in constraint-mediated algorithm design.

Beyond LCSS itself, our work highlights unexpected connections between subsequence and substring constraints, advancing the theoretical understanding of string problems and laying the foundation for new algorithmic techniques and complexity-theoretic insights in the rich space between classical string comparison paradigms.

2012 ACM Subject Classification Theory of computation → Problems, reductions and completeness

Keywords and phrases Longest Common Subsequence, Longest Common Substring, Conditional Lower Bound

Digital Object Identifier 10.4230/LIPIcs.CPM.2026.27

Funding *Shay Golan*: partially supported by Israel Science Foundation grant 810/21.
Matan Kraus: supported by the ISF grant no. 1926/19, by the BSF grant 2018364, and by the ERC grant MPM under the EU's Horizon 2020 Research and Innovation Programme (grant no. 683064).
Ely Porat: supported by the ISF grant no. 1926/19, by the BSF grant 2018364, and by the ERC grant MPM under the EU's Horizon 2020 Research and Innovation Programme (grant no. 683064).

1 Introduction and Previous Work

LCS. The *Longest Common Subsequence (LCS)* problem is one of the classical problems in computer science. Its first well-known quadratic-time dynamic programming solution was presented in 1974 by Wagner and Fischer [77]. In this problem, we are given two strings over the same alphabet and seek the length of the longest subsequence common to both strings, where a subsequence is obtained from a sequence by deleting zero or more elements. LCS has numerous applications, including in genetics and molecular biology [26, 30, 33, 80], optical character recognition (OCR) [42], and other domains. A wide range of algorithms has been proposed for solving LCS; see, for example, the surveys [21, 72].

Given its importance, many variants of LCS have been studied over the years. These include LCS for multiple strings (e.g., [1, 62, 79]), LCS under correlated sequence models [44], tree LCS (e.g., [11, 69]), weighted LCS (e.g., [10, 18]), the Longest Common Increasing Subsequence (e.g., [32, 27, 37]), constrained LCS (e.g., [13, 40, 46, 47, 57]), the Longest Common Parameterized Subsequence [45, 55, 63], and the Longest Common Palindromic Subsequence [16, 36, 61], among many other generalizations (e.g., [20, 22, 26, 54, 56]).

Considerable effort has also been devoted to identifying and exploiting structural properties of the input in order to design faster algorithms [14, 15, 38, 50, 53, 58, 70, 71, 83].

LCStr. In the *Longest Common Substring (LCStr)* problem, the goal is to find a common substring of the input strings, rather than a subsequence. Equivalently, LCStr can be viewed as a variant of the LCS problem in which the common subsequence is required to be a substring of **both** input strings. The LCStr problem is a fundamental problem in string processing, with numerous applications [4, 31, 35, 43]. It admits a straightforward linear-time solution using a (generalized) suffix tree [48, 81]. The problem has also been studied in various models and settings, including small-space algorithms [75, 65, 19], dynamic strings [8, 7, 9, 28], packed string representations [29], LCStr with up to k mismatches [41, 76, 64, 29], and LCStr with up to k edits [6].

To explore the gap between the LCS and LCStr problems, we briefly review their time complexity, approximation algorithms, and multi-string variants. This serves as a baseline for analyzing the LCSS problem.

Time complexity of LCS and its variations. Abboud et al. [1] and Bringmann and Künnemann [25] showed that any subquadratic-time algorithm for LCS would refute the Strong Exponential Time Hypothesis (SETH). Abboud et al. [2] further strengthened this result by showing that even a strongly subquadratic-time algorithm for LCS would refute a natural, weaker variant of SETH defined over branching programs.

Moreover, Bringmann and Künnemann [23] conducted a systematic study of the multivariate complexity of LCS, taking into account a wide range of parameters that influence the running time (such as input size, the length of the shorter string, and the length of the LCS). For any class of instances obtained by bounding each parameter by a polynomial in the input size, they established corresponding SETH-based lower bounds.

Generalizations of LCS and LCStr to multiple strings. To compute the *Longest Common Substring of k strings*, Hui [52] showed that a (generalized) suffix tree can be exploited to obtain a linear-time algorithm.

The problem of finding the LCS of k strings is commonly referred to as k-LCS. Maier [68] proved that k-LCS is NP-complete when k is unbounded, even over an alphabet of size 3. Over the years, several algorithms have been proposed for this problem. For example, Irving

and Fraser [62] presented an algorithm with running time $O(kn(n-\ell)^{k-1})$, where n is the common length of the input strings and ℓ is the length of an LCS. For the special case of three strings, Wang and Zhu [78] proposed two algorithms with running times $O(n^2\delta)$ and $O(n + R\log n + R^2)$, where δ denotes the minimum number of deletions required to obtain a common subsequence, and R is the number of matching triples (x, y, z) such that $S_1[x] = S_2[y] = S_3[z]$. In the worst case, however, these approaches require $\Omega(n^3) = \Omega(n^k)$ time.

Approximation versions of LCS and LCStr. For LCStr, the existence of a linear-time exact algorithm immediately implies a linear-time approximation algorithm. Moreover, a sublinear-time approximation with a constant factor is unlikely, as it would yield a sublinear-time algorithm for testing equality between two strings.

For LCS, simple near-linear-time algorithms achieve either a $1/|\Sigma|$-approximation or a $1/\sqrt{n}$-approximation, where n is the input size and Σ is the alphabet. In recent years, several works have studied the approximation of LCS. Abboud and Rubinstein [3] explored connections between approximate LCS and circuit complexity. Rubinstein and Song [74] showed that a $(1/2 + \epsilon)$-approximation can be obtained in subquadratic time for two equal-length strings over a binary alphabet. This result was later improved by Akmal and Vassilevska Williams [5], who achieved a $(1/2 + \epsilon)$-approximation in subquadratic time for any constant-size alphabet. For large alphabets, HajiAghayi et al. [49] improved the $1/n^{0.5}$-approximation to $1/n^{0.498}$ while maintaining linear running time. Subsequently, Bringmann et al. [24] further improved the approximation ratio to $1/n^{0.4}$ and introduced a trade-off between the approximation ratio and the running time. In an unpublished manuscript, Nosatzki [73] described a linear-time algorithm achieving an approximation ratio of $1/n^{o(1)}$. Notably, all algorithms whose approximation guarantees depend on n (rather than $|\Sigma|$) are randomized.

LCS variations bridging the gap between the LCS and LCStr problems. The contrasting complexity landscape of LCS and LCStr indicates that restricting the search to substrings significantly simplifies the problem. Several variants of LCS aim to bridge this gap by requiring the common subsequence to include substrings; see, e.g., [17, 20].

Recently, Banerjee et al. [17] introduced the *Longest Common Factor with Gaps* (k-LCFg) problem, in which the common subsequence is composed of at most k matching substrings, and the objective is to maximize their total length. They provide an algorithm with running time $\tilde{O}(nm^{1-1/3^{k-2}})$.

Benson et al. [20] defined the *LCSk* problem, where the common subsequence must consist of substrings of length exactly k in both input strings. Several algorithms have been proposed for *LCSk* [20, 34, 51, 84], all of which have quadratic worst-case running time.

Both problems incorporate substrings into the subsequence structure, but in different ways: k-LCFg allows substrings of unbounded length but limits their number, whereas *LCSk* allows an unbounded number of substrings, each of fixed length. This distinction becomes especially pronounced when $k = 1$: in this case, k-LCFg reduces to LCStr, while *LCSk* coincides with LCS.

Exploring the gap between LCS and LCStr. Recall that LCStr can be viewed as a variant of LCS in which the common subsequence is required to be a substring of **both** input strings. Given the substantial gap in complexity between LCS and LCStr (and their variants), a natural question arises: which additional constraints on LCS lead to problems whose complexity remains closer to that of LCS rather than LCStr?

Requiring substrings of fixed length (as in $LCSk$) appears to preserve the complexity characteristics of LCS. In contrast, allowing substrings of arbitrary length (as in k-LCFg) can break the quadratic-time barrier, bringing the problem closer to LCStr in terms of complexity.

Notably, both $LCSk$ and k-LCFg require the subsequence to consist of substrings from **both** input strings. This raises a natural question: what happens if this requirement is imposed on only **one** of the strings? To address this question, we consider an intermediate variant between LCS and LCStr, in which the common subsequence is required to be a substring of **one** of the input strings. This problem, called the *Longest Common Subsequence-Substring (LCSS)*, is defined as follows.

▶ **Problem 1** (LCSS). *Given two strings S and T over an alphabet Σ, of lengths n and m, respectively, compute the maximum length ℓ of a subsequence of S that is also a substring of T.*

For example, let $S = \text{cb}\boldsymbol{ab}\text{cbccb}\boldsymbol{aab}\text{ba}\boldsymbol{c}\text{aca}$ and $T = \boldsymbol{abaabc}\text{bac}$. Then $\mathsf{LCSS}(S, T) = 6$, and one longest common subsequence-substring is **abaabc**.

In addition to its theoretical interest, the LCSS problem is motivated by applications in cybersecurity. In such settings, malware signatures are often fragmented and transmitted across multiple packets, making detection challenging [66]. Given a set of signatures separated by a designated symbol, one may wish to identify the longest signature that appears as a subsequence in observed data, effectively detecting a single, possibly fragmented, occurrence. A related problem, *Dictionary Matching with a Few Gaps* [12], focuses on detecting all occurrences of patterns from a dictionary. In contrast, the LCSS formulation targets the detection of a single (longest) fragmented pattern.

Li, Deka, and Deka [67] introduced the LCSS problem and proposed a naive quadratic-time algorithm. Observe that the longest *prefix* of a string U that appears as a subsequence of S can be computed in $O(n)$ time using a two-pointer technique: the pointer on S advances at every step, while the pointer on U advances only upon a match. Applying this procedure to every suffix of T yields an $O(nm)$-time algorithm.

Our Contribution.

- We first show (in Section 3) that the LCSS problem has inherent quadratic-time complexity, up to subpolynomial factors, under the Strong Exponential Time Hypothesis (SETH), via a reduction from the Orthogonal Vectors problem. This lower bound indicates that the additional constraint imposed by LCSS, compared to LCS, does not lead to a computationally easier problem, and in particular does not bring its complexity closer to that of LCStr.
- We then show (in Section 4) that for instances where $\mathsf{LCSS}(S, T)$ is small, the running time can be improved to $O((n + m\ell)\log|\Sigma|)$, where $n = |S|$, $m = |T|$, $\ell = \mathsf{LCSS}(S, T)$, and Σ is the alphabet.
- Furthermore, we study (in Section 4.1) the LCSS problem for multiple strings (formally defined in Problem 15) in order to better understand its computational relationship to LCS and LCStr. Our results indicate that, in this setting, the problem behaves differently from both. In particular, for a constant number of strings $S_1, \ldots, S_{k_1}$ and $T_1, \ldots, T_{k_2}$, each of length n, we present an algorithm with running time $O(n\ell\log|\Sigma|)$, where $\ell = \mathsf{LCSS}(S_1, \ldots, S_{k_1}, T_1, \ldots, T_{k_2})$.
- Finally, we present (in Section 5) a $(1-\varepsilon)$-approximation algorithm for LCSS that runs in near-linear time, matching the time complexity of LCStr up to logarithmic factors. To the best of our knowledge, no comparable approximation results are known for the LCS problem.

2 Preliminaries

Below, we present some basic notations and a data structure definition used in this paper. For every two integers $i < j$ we denote by $[i..j] = \{i, i+1, i+2, \ldots, j\}$ the set of integers between i and j. We also denote $[i] = [1..i]$. A string S of length n over an alphabet $\Sigma = [|\Sigma|]$ (we assume that $|\Sigma|$ is polynomial in the input size) is a sequence of n symbols $S = S[1]S[2]\ldots S[n]$. The length of the string is denoted by $|S| = n$. For every $1 \leq i \leq j \leq n$, the *substring* $S[i..j] = S[i]S[i+1]\ldots S[j]$. If $i = 1$ we say that $S[i..j]$ is a prefix, and if $j = n$, it is a suffix. Given a string S of length n, the rth *cyclic shift* of S is the string $S[r+1..n] \cdot S[1..r]$ for some $r \in [1, n]$, where $\cdot$ denotes concatenation. Let S and T be two strings over the same alphabet. A common subsequence-substring of S and T (CSS) is a string X where X is a subsequence of S and a substring of T.

A useful tool for our results is the following data structure.

▶ **Lemma 2.** *For a string S of length n over an alphabet Σ, there is a linear-space data structure that preprocess S in $O(n \log |\Sigma|)$ time, and supports answering the following query in $O(\log |\Sigma|)$ time: Given a character $\sigma \in \Sigma$, and an index $i \geq 1$, determine the leftmost position of the character σ in $S[i..n]$ (or report $n+1$ if such an occurrence does not exist).*

Proof. We describe hereafter a data structure that satisfies the requirements of the lemma. In the preprocess we employ the following technique. S is divided into chunks C_f of length $|\Sigma|$ each (except for the last chunk which may be shorter). Chunk C_f represents substring $S[f|\Sigma|+1..(f+1)|\Sigma|]$ for $0 \leq f < n/|\Sigma|$. For each chunk C_f, we maintain two self-balancing binary search trees (BST), each of size $|\Sigma|$:

1. IN_{C_f}. For every $\sigma \in \Sigma$, $\mathsf{IN}_{C_f}[\sigma]$ stores a sorted array σOCC_{C_f} which stores the positions of all occurrences of σ within the corresponding substring of C_f. The arrays are constructed by scanning the substring corresponding to C_f and inserting each symbol into its respective array. We emphasize that for every σ the length of the array σOCC_{C_f}, is exactly the number of occurrences of σ in the substring corresponding to C_f.
2. OUT_{C_f}. For every $\sigma \in \Sigma$, $next_{C_f}[\sigma]$ stores the position of the first occurrence of σ that is after the end of chunk C_f, i.e., in $S[(f+1)|\Sigma|+1..n]$. If σ does not occur in the rest of S, then $next_{C_f}[\sigma] = n+1$. To fill the OUT_{C_f} values for each $\sigma \in \Sigma$ we scan the string S in reverse, from the end to the beginning, while maintaining an additional self-balancing binary search tree, *Last* which stores for each σ that we already scanned, its most recent occurrence. When we reach the end of a chunk's scope, i.e., position $(f+1)|\Sigma|$, we copy the contents of *Last* into OUT_{C_f}.

An example of the data structure constructed during preprocessing is shown in Figure 1.

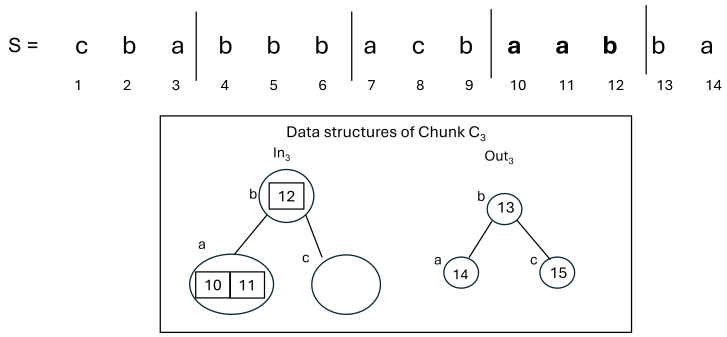

Figure 1 An example of the division of S into chunks of length $\Sigma = 3$ and the BST built for a single chunk.

Given a query of $\sigma \in \Sigma$, and an index $i \geq 1$, we need to find the first occurrence of σ in $S[i..n]$. We assume that $i \in [n]$, as otherwise the algorithm trivially returns $n+1$. Consider the chunk C_f where $i \in C_f$, i.e., $f = \lfloor i/|\Sigma| \rfloor$. We use $\mathsf{IN}_{C_f}[\sigma]$ to access σOCC_{C_f} on which we perform a binary search on σOCC_{C_f} to find the smallest element that is at least i. In case there is no such element, return the value saved in $\mathsf{OUT}_{C_f}[\sigma]$, or $n+1$ if such a value does not exist.

Time complexity. The construction of each BST over $|\Sigma|$ symbols requires $|\Sigma| \log |\Sigma|$ time. The time required to construct each σOCC_{C_f} is linear in its length, thus, all $O(|\Sigma|)$ σOCC_{C_f} arrays within a chunk C_f, collectively store $|\Sigma|$ symbols indices, are constructed in $O(|\Sigma| \log |\Sigma|)$ time. Given that there are $n/|\Sigma|$ chunks, the total time required for constructing the sorted arrays is $O(n \log |\Sigma|)$. The time complexity for the scan that the algorithm executes for filling the OUT arrays is also $O(n \log |\Sigma|)$.

Finally, to answer a query, the algorithm uses (at most) two balanced binary search tree of size at most $|\Sigma|$ and executes a binary search on an array of length $|\Sigma|$. Thus, the time complexity of answering a query is $O(\log |\Sigma|)$ ◀

Approximation Algorithm. Let $\mathcal{P}$ be a maximization problem, and let $\mathrm{OPT}(I)$ denote the optimal value of an instance I. An algorithm $\mathcal{A}$ is said to be a $(1-\varepsilon)$-approximation for $\mathcal{P}$ (for $\varepsilon > 0$) if for every instance I, it returns a solution of value $\mathcal{A}(I)$ such that $(1-\varepsilon)\mathrm{OPT}(I) \leq \mathcal{A}(I) \leq \mathrm{OPT}(I)$. That is, the value of the solution returned by $\mathcal{A}$ is guaranteed to be at least a $(1-\varepsilon)$-fraction of the optimal value.

Suffix Tree Construction. The following lemma by Farach-Colton, Ferragina and Muthukrishnan [39] introduce a linear-time construction algorithm for a suffix tree of a string with polynomial integer alphabet.

▶ **Lemma 3** (Farach-Colton et al. [39]). *Let S be a string of length n over an integer alphabet of size $n^{O(1)}$. Then one can construct the suffix tree of S in $O(n)$ time on the word-RAM model.*

Given a set of strings over an alphabet of size polynomial in the total input length, all suffix trees can be constructed in linear time, as stated in the following lemma. This will be useful in our algorithms.

▶ **Lemma 4.** *Let $\mathcal{T} = \{T_1, \ldots, T_{k_T}\}$ be a set of non-empty strings over an integer alphabet $\Sigma = \{1, 2, \ldots, |\Sigma|\}$. For every $i \in [k_T]$, let $|T_i| = m_i$, and let $n = \sum_{i=1}^{k_T} m_i$. If $|\Sigma| \leq n^c$ for some constant c, then on the word-RAM model there exists an algorithm that computes the suffix tree of every string in $\mathcal{T}$ in total $O(n)$ time.*

Proof. The proof consists of two steps.

First, we rename the letters of each string so that each string gets its own alphabet of size at most its length. For every occurrence of a letter in the input, create a record (i, a, p), where i is the index of the string, $a = T_i[p]$, and p is the position of this occurrence inside T_i. There are exactly n such records.

We sort these records lexicographically by the pair (i, a). Since $i \in [k_T] \subseteq [n]$ and $a \in [|\Sigma|] \subseteq [n^c]$, both coordinates fit in $O(\log n)$ bits. Hence the sorting can be done in $O(n)$ time by radix sort.

We now scan the sorted list. For each fixed string T_i, whenever we encounter a new letter value a, we assign it the next rank in $\{1, 2, \ldots\}$ among the distinct letters of T_i. Let T_i' be the resulting renamed string. This scan clearly takes $O(n)$ time. If σ_i is the number of

distinct letters in T_i, then T'_i is over the alphabet $\{1, 2, \ldots, \sigma_i\}$, where $\sigma_i \leq m_i$. Moreover, the renaming preserves equality and inequality of letters inside each fixed string, so the suffix tree of T_i is obtained from the suffix tree of T'_i by replacing the renamed letters by the original ones. Thus it is enough to construct the suffix trees of the strings $T'_1, \ldots, T'_{k_T}$.

Second, for every $i \in [k_T]$, append to T'_i a terminal symbol $\$_i$ that is smaller than all alphabet symbols and appears only once. Since we build each suffix tree separately, we may simply use the same fresh terminal symbol $\$_i = 0$ for every i. Now $T'_i 0$ has length $m_i + 1$ and its alphabet size is at most $\sigma_i + 1 \leq m_i + 1$. Therefore, by Lemma 3, the suffix tree of $T'_i 0$ can be built in $O(m_i)$ time.

Summing over all $i \in [k_T]$, the total time for this step is $\sum_{i=1}^{k_T} O(m_i) = O(n)$. Together with the $O(n)$ time of the renaming step, the overall running time is $O(n)$. Hence all suffix trees can be computed in total $O(n)$ time. ◀

3 LCSS Lower Bound

In this section, we show that a quadratic complexity is inherently necessary for solving the LCSS problem, similar to the LCS problem. Formally, we prove the following theorem.

▶ **Theorem 5.** *Let $\varepsilon > 0$. There is no algorithm for the LCSS problem with $O((nm)^{1-\varepsilon})$ running time, unless SETH is false.*

We develop a reduction from the *Orthogonal Vectors* (OV) problem. Williams [82] shows that any algorithm for the OV problem with strongly sub-quadratic running time would break the Strong Exponential Time Hypothesis (SETH), introduced in [59, 60]. Formally, we define the OV problem, and state its conditional lower bound as follows.

▶ **Problem 6** (OV). *Given two sets $A, B \subseteq \{0,1\}^d$ with $|A| = |B| = n$, determine whether there exist $\mathbf{x} \in A$, $\mathbf{y} \in B$ such that $\mathbf{x} \cdot \mathbf{y} = 0$, where $\mathbf{x} \cdot \mathbf{y} = \sum_{i=1}^{d} \mathbf{x}[i] \cdot \mathbf{y}[i]$.*

▶ **Lemma 7** ([82]). *If for some $\varepsilon > 0$, OV on n vectors in $\{0,1\}^d$ for $d = O(\log n)$ can be solved in $O(n^{2-\varepsilon} \cdot \mathrm{poly}(d))$ time, then SETH is false.*

Let $\mathbf{0}$ denote the 0^d vector and $\mathbf{1}$ denote the 1^d vector. We note that instances of the OV problem where $\mathbf{0} \in A \cup B$ are easy to solve in linear time. Therefore, when reducing the OV problem to other problems, we assume without loss of generality that $\mathbf{0} \notin A \cup B$.

For two strings S and T we say that T is a cyclic subsequence of S if and only if there is an $r \in [|T|]$ such that the rth cyclic shift of T is a subsequence of S. We define an intermediate problem, the Cyclic Subsequence problem, as follows.

▶ **Problem 8** (Cyclic Subsequence). *Given two input strings S and T of lengths n and m respectively, return true if and only if T is a cyclic subsequence of S.*

We will first show (in Section 3.1) a reduction from the OV problem to the Cyclic Subsequence problem, establishing the hardness of the Cyclic Subsequence problem. Then (in Section 3.2), we describe a second reduction from the Cyclic Subsequence problem to the LCSS problem.

3.1 Reducing the OV Problem to Cyclic Subsequence Using Constant Size Alphabet

In this section, we prove a conditional lower bound for Problem 8, by presenting a reduction from the OV problem, using a constant size alphabet.

The Reduction. Given two sets $A = \{\mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_n\} \subseteq \{0,1\}^d$ and $B = \{\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n\} \subseteq \{0,1\}^d$ we define strings SG and TG, over an alphabet $\Sigma = \{\psi, \#, \$\}$ as follows. For $\mathbf{u} \in A$, let $x_i = \psi$ for $i \in [d]$ such that i is a 0-entry index in $\mathbf{u}$, otherwise x_i is an empty string. Similarly, for $\mathbf{v} \in B$, let $y_i = \psi$ for $i \in [d]$ such that i is a 1-entry index in $\mathbf{v}$, otherwise y_i is an empty string. We define the vector gadgets (where $\bigodot$ denotes sequential concatenation):

$$\mathsf{VG}_0(\mathbf{u}) = \left(\bigodot_{i\in[d]} (x_i\#)\right)\$. \qquad \mathsf{VG}_1(\mathbf{v}) = \left(\bigodot_{i\in[d]} (y_i\#)\right)\$$$

The following lemma establishes the relation between the orthogonality of $\mathbf{u}$ and $\mathbf{v}$ and the property of one vector gadget being a subsequence of the other.

▶ **Lemma 9.** *$\mathsf{VG}_1(\mathbf{v})$ is a subsequence of $\mathsf{VG}_0(\mathbf{u})$ if and only if $\mathbf{v} \cdot \mathbf{u} = 0$*

Proof. According to the construction, $\mathsf{VG}_1(\mathbf{v})$ consists of d occurrences of # symbols, with a ψ preceding the ith # whenever $\mathbf{v}[i] = 1$. Similarly, $\mathsf{VG}_0(\mathbf{u})$ consists of d occurrences of # symbols, with a ψ preceding the ith # whenever $\mathbf{u}[i] = 0$.

If the vectors $\mathbf{v}$ and $\mathbf{u}$ are orthogonal, then for every index i such that $\mathbf{v}[i] = 1$, it holds that $\mathbf{u}[i] = 0$. Thus, in addition to the d matching # symbols, if a ψ symbol appears before the ith # in $\mathsf{VG}_1(\mathbf{v})$ then a corresponding ψ symbol appears before the ith # in $\mathsf{VG}_0(\mathbf{u})$. Finally, note that the \$ symbols always match, as they mark the ends of both vector gadgets. Therefore, $\mathsf{VG}_1(\mathbf{v})$ is a subsequence of $\mathsf{VG}_0(\mathbf{u})$.

If $\mathsf{VG}_1(\mathbf{v})$ is a subsequence of $\mathsf{VG}_0(\mathbf{u})$, this implies that for every ψ symbol in $\mathsf{VG}_1(\mathbf{v})$, if it appears before the ith # in $\mathsf{VG}_1(\mathbf{v})$, then a corresponding ψ must appear before the ith # in $\mathsf{VG}_0(\mathbf{u})$, as otherwise not all # symbols in $\mathsf{VG}_1(\mathbf{v})$ could be matched with those in $\mathsf{VG}_0(\mathbf{u})$. Therefore, for every index $i \in [d]$ such that $\mathbf{v}[i] = 1$, it must be that $\mathbf{u}[i] = 0$. This implies that the vectors $\mathbf{u}$ and $\mathbf{v}$ are orthogonal. ◀

Given A and B we define sequences gadgets SG and TG, as follows.

$$\mathsf{SG} = \bigodot_{j\in[n]} \left(\mathsf{VG}_0(\mathbf{u}_j)\mathsf{VG}_0(\mathbf{0})\right)$$

$$\mathsf{TG} = \left(\bigodot_{i\in[n]} \mathsf{VG}_1(\mathbf{v}_i)\right) \mathsf{VG}_1(\mathbf{1})$$

A visualization of the defined sequence gadgets is shown in Figure 2. We refer to $\mathsf{VG}_0(\mathbf{u}_j)\mathsf{VG}_0(\mathbf{0})$ as the j-th segment of SG.

Vector-shifts. Recall that TG ends with a \$ sign. We define a *vector-shift* on TG as the operation of applying (left) cyclic shifts until TG ends with a \$ sign again. Notice that TG encodes the sequence $(\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n, \mathbf{1})$, and after a single vector-shift we obtain an encoding of the sequence $(\mathbf{v}_2, \ldots, \mathbf{v}_n, \mathbf{1}, \mathbf{v}_1)$. In general, if a string is the encoding of a sequence of vectors, then by applying a vector-shift on the string we obtain the encoding of cyclic shifted sequence of vectors.

The following two lemmas prove that TG is a cyclic subsequence of SG if and only if there are vectors $\mathbf{u} \in A$ and $\mathbf{v} \in B$ such that $\mathbf{u} \cdot \mathbf{v} = 0$.

▶ **Lemma 10.** *If there are vectors $\mathbf{u}_j \in A$ and $\mathbf{v}_i \in B$ such that $\mathbf{u}_j \cdot \mathbf{v}_i = 0$ then TG is a cyclic subsequence of SG.*

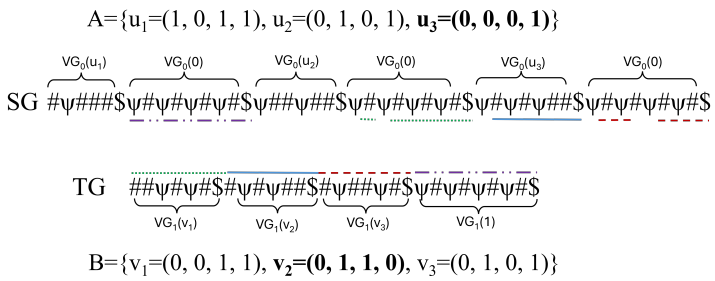

Figure 2 An example of the reduction construction for $d = 4$, $n = 3$. The dotted lines represent the corresponding matched symbols. In the example $\mathbf{u}_3$ is orthogonal to $\mathbf{v}_2$ and the $LCSCS(\mathsf{SG}, \mathsf{TG}) = |\mathsf{TG}|$.

Proof. Since we are searching for a cyclic subsequence, TG can be vector-shifted until the obtained string is the encoding of cyclic shift of $(\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n, \mathbf{1})$ such that $\mathbf{v}_i$ appears in the jth position of the sequence. Let TG' be this cyclic shift of TG

Since $\mathbf{u}_j \cdot \mathbf{v}_i = 0$, Lemma 9 implies that $\mathsf{VG}_1(\mathbf{v}_i)$ is a subsequence of $\mathsf{VG}_0(\mathbf{u}_j)$. Now, consider the pair of prefixes and the pair of suffixes of SG and TG' until and after $\mathsf{VG}_1(\mathbf{v}_i)$ and $\mathsf{VG}_0(\mathbf{u}_j)$. In SG, to the left of $\mathsf{VG}_0(\mathbf{u}_j)$, there are $j-1$ occurrences of $\mathsf{VG}_0(\mathbf{0}) = (\psi\#)^d\$$. Similarly, in TG' there are $j-1$ vector gadgets of type VG_1 to the left of $\mathsf{VG}_1(\mathbf{v}_i)$, each of which is a subsequence of $\mathsf{VG}_0(\mathbf{0})$, by Lemma 9. Similarly, to the right of $\mathsf{VG}_0(\mathbf{u}_j)$ in SG, there are $n+1-j$ occurrences of $\mathsf{VG}_0(\mathbf{0})$, these include $n-j$ occurrences from the last $n-j$ segments of SG, along with one additional occurrence from the jth segment. Correspondingly, in TG' there are $n-j+1$ vector gadgets of type VG_1 to the right of $\mathsf{VG}_1(\mathbf{v}_i)$, each of which is a subsequence of $\mathsf{VG}_0(\mathbf{0})$, by Lemma 9. To conclude, we deduce that TG', that is a cyclic shift of TG, is a subsequence of SG, hence, we prove the lemma. ◀

▶ **Lemma 11.** *If* TG *is a cyclic subsequence of* SG *then there are vectors* $\mathbf{u} \in A$ *and* $\mathbf{v} \in B$ *such that* $\mathbf{u} \cdot \mathbf{v} = 0$.

Proof. Let TG' be a cyclic shift of TG such that TG' is a subsequence of SG. First, we show that, without loss of generality, we can assume that TG' is a result of applying vector-shifts on TG. If this is not the case, it means that there is a vector gadget $\mathsf{VG}_1(\mathbf{v}_i)$ for some $\mathbf{v}_i \in B \cup \{\mathbf{1}\}$ that was split by the cyclic shifts. Let r' be the first index in TG' such that $\mathsf{TG}'[r'] = \$$ (this is the \$ sign of $\mathsf{VG}_1(\mathbf{v}_i)$). Since TG' is a subsequence of SG we can assume (without loss of generality) that the prefix of $\mathsf{VG}_1(\mathbf{v}_i)$ was matched as a subsequence to the end of SG, specifically, to the last occurrence of $\mathsf{VG}_0(\mathbf{0}) = (\psi\#)^d\$$. As a result, we can perform a cyclic shift of r' symbols, meaning the prefix $\mathsf{TG}'[1..r']$ is appended to the end of the string and removed from the beginning, to obtain a new string TG'' that ends with the complete vector gadget $\mathsf{VG}_1(\mathbf{v}_i)$, and therefore is a result of vector-shifts on TG. The complete $\mathsf{VG}_1(\mathbf{v}_i)$ can still be matched as a subsequence to the final occurrence of $\mathsf{VG}_0(\mathbf{0}) = (\psi\#)^d\$$ of SG. Hence, if TG' is matched as a subsequence to SG, then TG'' matches as well.

We say that μ is a *valid* one to one mapping from $[|\mathsf{TG}'|]$ if for every $x \in [|\mathsf{TG}'|]$, $\mu(x) = y$ where $\mathsf{TG}'[x] = \mathsf{SG}[y]$ and $\mu(x) < \mu(x+1)$ for every $x \in [n-1]$. Given that TG' is a subsequence of SG, there exists at least one valid mapping μ.

We say that an instance $\mathsf{VG}_0(\mathbf{u})$ *μ-touches* an instance $\mathsf{VG}_1(\mathbf{v})$ if a symbol from $\mathsf{VG}_1(\mathbf{v})$ is mapped by μ to a symbol from $\mathsf{VG}_0(\mathbf{u})$ (there exists $x \in [|\mathsf{TG}'|]$ such that $\mu(x) = y$, where $\mathsf{TG}'[x]$ is part of $\mathsf{VG}_1(\mathbf{v})$ for some $\mathbf{v} \in B \cup \{\mathbf{1}\}$ and $\mathsf{SG}'[y]$ is part of $\mathsf{VG}_0(\mathbf{u})$ for some $\mathbf{u} \in A \cup \{\mathbf{0}\}$).

▷ Claim 12. Let $\mathsf{SG}[a..b]$ be a substring of SG such that $\mathsf{SG}[a..b] = \mathsf{VG}_0(\mathbf{u})$ for some $\mathbf{u} \in A \cup \{\mathbf{0}\}$. Then, $\mathsf{SG}[a..b]$ μ-touches at most one vector gadget (of type VG_1) in TG'.

Proof. Intuitively, the claim holds since $\mathsf{SG}[a..b]$ ends with a \$, and there is at least one \$ between every two vector gadgets in TG'. Formally, assume to the contrary that $\mathsf{SG}[a..b]$ μ-touches two vector gadgets of TG'. In particular, let a' and b' be the leftmost and rightmost indices of symbols of TG' that are mapped to some index in $[a..b]$, respectively (for example see Figure 3).

By definition, $\mathsf{TG}'[a'..b']$ is a subsequence of $\mathsf{SG}[a..b]$. Since $\mathsf{SG}[a..b]$ μ-touches two vector gadgets of TG' it must be that the rightmost index z' of the vector gadget that contains a' in TG' has $z' < b'$. In addition, since z' is the last index of a vector gadget, it holds that $\mathsf{TG}'[z'] = \$$. Recall that $\mathsf{SG}[a..b]$ contains exactly one \$ sign, and this \$ sign is at position b of SG. Thus, it must be that $\mu(z') = b$. However, by the monotonicity of μ, it must be that $\mu(b') \geq \mu(z'+1) > b$, which is a contradiction. ◁

Notice that it can be the case that several valid mappings exist. Consider a mapping μ' which maximize the number of $\mathsf{VG}_0(\mathbf{0})$ gadgets that μ'-touches a vector gadget of TG'. Due to the reduction construction, SG contains n occurrences of $\mathsf{VG}_0(\mathbf{0})$ and TG' contains $n+1$ vector gadgets, each a subsequence of $\mathsf{VG}_0(\mathbf{0})$, by Lemma 9. By Claim 12 and the pigeonhole principle, there is at least one vector gadget VG_1 in TG that is not μ'-touched by any instance of $\mathsf{VG}_0(\mathbf{0})$. Let $\mathsf{TG}'[a'..b']$ be this vector gadget in TG'. Since μ' is a valid mapping, $\mathsf{TG}'[a'..b']$ must be touched by at least one vector gadget of SG. Assume to the contrary that $\mathsf{TG}'[a'..b']$ is touched by more than one vector gadget of SG. Let $a = \mu'(a')$ and $b = \mu'(b')$. Recall that $\mathsf{TG}'[a'..b']$ is not touched by a $\mathsf{VG}_0(\mathbf{0})$ vector gadget. Thus, by the reduction construction, it must be that $\mathsf{SG}[a..b]$ contains a $\mathsf{VG}_0(\mathbf{0})$ vector gadget. Thus, there exists a valid mapping μ'' such that this $\mathsf{VG}_0(\mathbf{0})$ vector gadget is also μ''-touch: by modifying μ' such that $\mathsf{SG}[a..b]$ is mapped to this $\mathsf{VG}_0(\mathbf{0})$ vector gadget (such a mapping exist by Lemma 9). Contradicting the maximality of μ'. Thus, it must be the case that $\mathsf{TG}'[a'..b']$ is touched by a single vector gadget of SG. This vector gadget must be $\mathsf{VG}_0(\mathbf{u})$ for some $\mathbf{u} \in A$. Notice that $\mathsf{TG}'[a'..b']$ cannot be $\mathsf{VG}_1(\mathbf{1})$ since $\mathbf{u} \neq \mathbf{0}$ by definition of A.

Hence, if TG' is a subsequence of SG, at least one $\mathsf{VG}_1(\mathbf{v}_i)$ for some $\mathbf{v}_i \in B$ must be a subsequence of $\mathsf{VG}_0(\mathbf{u}_j)$ for some $\mathbf{u}_j \in A$, implying $\mathbf{v}_i, \mathbf{u}_j$ are orthogonal, according to Lemma 9. ◀

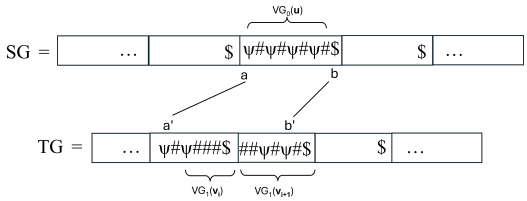

Figure 3 An example for Claim 12. Each rectangle represents a vector gadget. The lines indicate the $\mu-$touching boundaries of elements from $\mathsf{VG}_0(\mathbf{u})$.

We are now ready to prove the hardness of Problem 8.

▶ **Lemma 13.** *For every $\varepsilon > 0$, there is no algorithm that solves Problem 8 in $O((nm)^{1-\varepsilon})$ time, unless SETH is false.*

Proof. Assume that Problem 8 can be solved in $O((nm)^{1-\varepsilon})$ time for some $\varepsilon > 0$. Let A and B be two sets of size N, which are an input for the OV problem (Problem 6). The reduction described above takes $O(dN)$ time to construct SG and TG, both of size $O(dN)$.

By Lemmas 10 and 11, TG is a cyclic-subsequence of SG if and only if there are vectors $\mathbf{u} \in A$ and $\mathbf{v} \in B$ such that $\mathbf{u} \cdot \mathbf{v} = 0$. Therefore, there is an algorithm that solves Problem 6 in $O(dN + (dN \cdot dN)^{1-\varepsilon}) = O(N^{2-\varepsilon} \cdot \text{poly}(d))$ time, which, by Lemma 7, refutes SETH. ◀

3.2 Reducing Cyclic Subsequence to LCSS

After establishing the hardness of the Cyclic Subsequence problem (Problem 8), we are finally ready to prove the hardness of the LCSS problem by presenting a reduction from Problem 8 to the LCSS problem.

The Reduction. Given an input for the Cyclic Subsequence problem, we construct two new strings $S' = S$, and $T' = T \cdot T$.

▶ **Lemma 14.** $\mathsf{LCSS}(S', T') \geq |T|$ *if and only if* T *is a cyclic subsequence of* S.

Proof. If T is a cyclic subsequence of S, then there is an $r \in [|T|]$ such that the rth cyclic shift of T, i.e., $T[r+1..|T|] \cdot T[1..r]$, is a subsequence of S. Therefore, $T'[r+1..r+|T|]$ is a subsequence of S' and so $\mathsf{LCSS}(S', T') \geq |T|$.

On the other hand, if $\mathsf{LCSS}(S', T') \geq |T|$ then there is an index $r \in [|T|]$ such that $T'[r+1..r+|T|]$ is a subsequence of S'. Therefore, the rth cyclic shift of T is a subsequence of S and so T is a cyclic subsequence of S. ◀

Lemmas 13 and 14 yield the proof of Theorem 5.

Proof of Theorem 5. Assume that there is an algorithm $\mathcal{A}$ that computes LCSS in time $O((nm)^{1-\varepsilon})$ for some $\varepsilon > 0$. Given an input pair S and T of lengths N and M, respectively, to the Cyclic Subsequence problem, we construct S' and T' of lengths $n = N$ and $m = 2M$, respectively, as described above, in $O(N + M)$ time. By Lemma 14, running $\mathcal{A}$ on S' and T' returns at least $|T|$ if and only if If T is a cyclic subsequence of S. Therefore, there exists an algorithm that solves Cyclic Subsequence in $O(N + M + (NM)^{1-\varepsilon})$ time. By Lemma 13 this refutes SETH. ◀

4 LCSS Problem for Multiple Strings

We now turn our attention to a generalized version of the LCSS problem, involving multiple input strings, the Longest Common Subsequence-Substring problem for Multiple Strings ((k_S, k_T)-LCSS) problem, hereafter defined (see Figure 4 for an example).

▶ **Problem 15** ((k_S, k_T)-LCSS). *Given two sets of input strings* $\mathcal{S} = \{S_1, \ldots, S_{k_S}\}$ *and* $\mathcal{T} = \{T_1, \ldots, T_{k_T}\}$ *over an alphabet* Σ*, where the strings of* $\mathcal{S}$ *are of lengths* $n_1, \ldots, n_{k_S}$ *and the strings of* $\mathcal{T}$ *are of lengths* $m_1, \ldots, m_{k_T}$*. The goal is to find the length of the longest string* X *such that the following conditions hold:*

1. X *is a subsequence of each* $S \in \mathcal{S}$*.*
2. X *is a substring of each* $T \in \mathcal{T}$*.*

This length is denoted as $\mathsf{LCSS}(\mathcal{S}, \mathcal{T}) = |X|$.

We begin with a solution to the standard LCSS problem, i.e., (1,1)-LCSS. The algorithm iterates over each suffix of T, $T[i..m]$ for $1 \leq i \leq m$, and searches for its longest prefix that occurs as a subsequence in S. This is done by sequentially locating the symbols of the current suffix of T in S, with the only constraint being that the corresponding symbols must appear in S in increasing order. The matching process terminates when the current symbol of T is no longer found in the remaining portion of S. Since each iteration considers a different

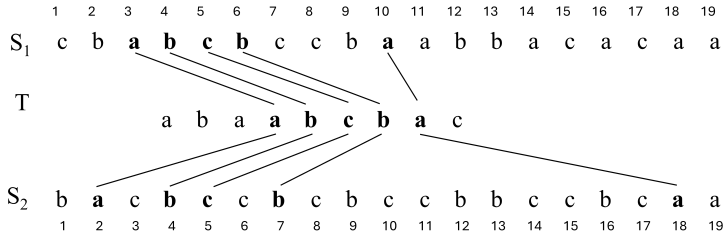

Figure 4 An example of the (2,1)-LCSS. The (2,1)-$\mathsf{LCSS}(S,T)$ = abcba with $\ell = 5$.

suffix of T independently of the others, a variable is maintained to store the maximum length of the obtained common subsequence-substring so far. The LCSS algorithm is outlined in Algorithm 1.

Algorithm 1 $\mathsf{LCSS}(S,T)$.

```
1  max ← 0;
2  Initialize the data structure described in Lemma 2, denoted by D, for S;
3  for i ← 1 to m do
4  |  j ← 1 ; a ← 0;
5  |  while i + a ≤ m do
6  |  |  h ← first occurrence of T[i + a] in S[j ... n] using D;
7  |  |  if h ≠ n + 1 then
8  |  |  |  j ← h + 1; a ← a + 1;
9  |  |  else
10 |  |  |  break;
11 |  max ← max{a, max};
12 return max;
```

▶ **Theorem 16.** *Algorithm 1 solves the LCSS Problem in $O((n + m\ell)\log|\Sigma|)$ time and using $O(n + m)$ space where n and m are the lengths of strings S and T respectively and $\ell = \mathsf{LCSS}(S,T)$.*

Proof. We begin with the correctness of the algorithm. We first note that the value of a at any time is a length of some CSS of S and T, and therefore the output of the algorithm is at most $\mathsf{LCSS}(S,T)$. On the other hand, let $T[i..i+\ell-1]$ be a longest CSS. Consider the ith iteration of line Line 3. It is straightforward to see that by choosing greedily the first valid occurrence of a character at any inner iteration of Line 5, the algorithm obtains the longest possible prefix of $T[i..m]$ that is a subsequence of S. Therefore, the algorithm finds in the ith iteration of Line 3 a CSS of length at least ℓ. Thus, the output of the algorithm is exactly $\mathsf{LCSS}(S,T)$.

Regarding the complexities, the initialization of the data structure of Lemma 2 at Line 1 takes $O(n\log|\Sigma|)$ time. It is clear that any outer iteration of Line 3 executes at most $\ell+1$ iterations of the inner-loop of Line 5. Each iteration of Line 5 takes $O(\log|\Sigma|)$ time, dominated by the invocation of the query on D. Thus, the time complexity of the algorithm is $O((n + m\ell)\log|\Sigma|)$. The space requirement derives from Lemma 2. ◀

4.1 (k_S, k_T)-LCSS Solution

In the general case of (k_S, k_T)-LCSS, the input includes a set $\mathcal{S}$ of k_S strings, and a set $\mathcal{T}$ of k_T strings. The goal is to find the (length of the) longest string that is a subsequence of all the strings in $\mathcal{S}$ and also a substring of all the strings in $\mathcal{T}$.

We follow the idea of Algorithm 1, that is - the algorithm uses one string T from $\mathcal{T}$ as an anchor, and consider every suffix of T separately. For a suffix $T[i..|T|]$, the algorithm finds the largest value $\ell = \ell_i$ such that the prefix of $T[i..|T|]$ of length ℓ (which is $T[i..i+\ell-1]$) is both a substring of all strings in $\mathcal{T}$ and a subsequence of all strings in $\mathcal{S}$. Notice that $\ell = \mathsf{LCSS}(\mathcal{S}, \mathcal{T}) = \max\{\ell_i \mid i \in [|T|]\}$. Thus, from now on we focus on the computation on a specific suffix $T[i..|T|]$.

To make this computation, the algorithm first computes in linear time for every suffix $T[i..|T|]$ the largest ℓ such that $T[i..i+\ell-1]$ is a substring of all strings of $\mathcal{T}$ (ignoring the requirement to be a subsequence of strings in $\mathcal{S}$). This is possible due to lemma Lemma 17. For its proof we utilize suffix trees. A suffix tree, first introduced by Weiner [81], is a compressed trie that represents all the suffixes of a given string. Each edge is labeled with a substring of the input string, and every path from the root to a leaf corresponds to a distinct suffix of the string. Edges can be stored using the starting position and length of the corresponding substring in the input string, resulting in a total space usage that is linear in the length of the string.

▶ **Lemma 17.** *Let $\mathcal{T}$ be a set of k_T strings of lengths $m_1, m_2, \ldots, m_{k_T}$ and let T be a string of length m such that $m \leq m_i$ for every $i \in [k_T]$. There exists an algorithm that outputs for every $i \in [m]$ the largest ℓ_i such that $T[i..i+\ell_i-1]$ is a substring of all the strings of $\mathcal{T}$. The algorithm runs in $O(\sum_{j=1}^{k_T} m_j)$ time.*

Proof. We suggest the following algorithm yielding an array of length m, *MaxCS*, where for every $i \in [m]$, $MaxCS[i]$ stores the length of the longest prefix of $T[i..n]$ that is a common substring of all strings in $\mathcal{T}$, and is initialized to ∞.

For every $T_k \in \mathcal{T}$ let $T'_k = T\$_1 T_k \$_2$ where $\$_1, \$_2$ are unique terminal symbols. Let $\mathcal{T}' = \{T'_k \mid k \in [k_T]\}$ be the set of all strings T'_k. We use Lemma 4 to compute all suffix trees of strings in $\mathcal{T}'$, and for every k we denote by ST_k the suffix tree of $T'_k = T\$_1 T_k \$_2$.

The algorithm processes each suffix tree ST_k as follows. Recall that every leaf in the suffix tree corresponds to a suffix of the input string. Each leaf that corresponds to an index from the string T is marked by 1, and each leaf that corresponds to an index from the string T_k is marked by 2. Each leaf marked by 1 stores the starting index of the suffix it represents. Additionally, for each node v, we use $len(v)$ to denote the length of the string represented by the path from the root to v.

We associate to each $v \in ST_k$: (1) a boolean flag $sec(v)$ indicating whether the string represented by the path from the root to v is a substring of a suffix of T_k, initialized to *false*. (2) a pointer $up(v)$, pointing to the nearest ancestor of v that represents the longest prefix of a suffix of T_k.

To update the flags $sec(v)$, we traverse ST_k in postorder. A leaf node v marked with 2 sets its flag $sec(v)$ to *true*. For each internal node $v \in ST_k$, if any of its children v' satisfies $sec(v') = true$ then $sec(v)$ is also set to *true*. Next, we perform an additional preorder traversal of ST_k. During this traversal, each node v is assigned a value $up(v)$ as follows: if $sec(v) = true$, then $up(v) = v$, otherwise, $up(v) = up(u)$ where u is the parent of v.

We iterate over all indices $i \in [m]$. For each index i let v be the leaf v representing the suffix $T[i..m]$ in ST_k. Notice that $len(up(v))$ indicates the length of the longest prefix of $T[i..m]$ that is also a prefix of some suffix of T_k. We update $MaxCS[i] \leftarrow \min(MaxCS[i], len(up(v)))$. We apply this procedure for each $T_k \in \mathcal{T}$. After completing all iterations, we set $\ell_i \leftarrow MaxCS[i]$.

Correctness. For each suffix of T, obtaining its longest common prefix with some $T_k \in \mathcal{T}$ yields the longest prefix of $T[i..m]$ which is a substring of T_k. The length of this substring is given by $len(up(v))$, where v is the node representing the suffix $T[i..m]$ in ST_k.

By maintaining $MaxCS[i]$ as the minimum of all such values across every $T_k \in \mathcal{T}$, we ensure that at the end of the procedure, $MaxCS[i]$ stores the length of the longest prefix of $T[i..m]$ that is a substring of all strings in $\mathcal{T}$, as required.

Complexities. Notice that each string $T'_k \in \mathcal{T}'$ is of length $|T| + |T_k| + 2 = m + m_k + 2 \leq 2m_k + 2 = O(m_k)$. Hence, the algorithm of Lemma 4 takes $O\left(\sum_{k=1}^{k_T} m_k\right)$ time. For each iteration over $T_k \in \mathcal{T}$ updating the necessary values including those in $MaxCS$ array for each $T_k \in \mathcal{T}$, takes linear time in the size of ST_k, i.e., $O(m + m_k) = O(m_k)$ time. Therefore, the overall time complexity of the algorithm is $O\left(\sum_{k=1}^{k_T} m_k\right)$. ◀

Let $\hat{\ell}$ be the output of Lemma 17 for index i. When the algorithm considers the suffix $T[i..|T|]$, it needs to find the largest $\bar{\ell}$ such that $T[i..i+\bar{\ell}-1]$ is a subsequence of all strings in $\mathcal{S}$. Notice that $\ell = \ell_i$ is exactly $\ell = \min\{\hat{\ell}, \bar{\ell}\}$. Similar to Algorithm 1, the algorithm advances a pointer a on $T[i..|T|]$, verifying at every iteration that $T[i..i+a]$ is still a subsequence of all strings of $\mathcal{S}$ (notice that now instead of verifying this property for a single string S, we need to make the verification for all of the strings in $\mathcal{S}$ together). However, the algorithm does not need to advance a more than $\hat{\ell}$ times, since $\ell = \min\{\hat{\ell}, \bar{\ell}\}$.

So, it only remains to describe how the algorithm is advancing the counter a while verifying that $T[i..i+a-1]$ is a subsequence of all strings of $\mathcal{S}$, instead of a single string S (see Algorithm 2). To do this, at the initialization, the algorithm constructs the data structure of Lemma 2 for each string $S_s \in \mathcal{S}$ separately, denoted as D_s. When reading the suffix $T[i..|T|]$, the algorithm initializes for every $s \in [k_S]$ a pointer j_s that maintains at the beginning of every iteration of Line 9 the smallest index such that $T[i..i+a-1]$ is a subsequence of $S_s[1..j_s-1]$. Before advancing a to $a+1$, the algorithm makes the verification for every $s \in [k_S]$ one after the other. The algorithm stops advancing a when either $a = \hat{\ell}$ or there is an $s \in [k_S]$ such that $T[i..i+a]$ is not a subsequence of S_s, which the algorithm identifies by the fact that the data structure D_s reports $n_s + 1$.

Algorithm 2 multiple-LCSS$(\mathcal{S}, \mathcal{T})$.

```
1  max ← 0;
2  T ← a shortest string of 𝒯;
3  (ℓ̂_1, ℓ̂_2, ..., ℓ̂_|T|) ← run the algorithm of Lemma 17 for T;
4  For all s ∈ [k_S], D_s ← initialize the data structure of Lemma 2 for S_s;
5  for i ← 1 to |T| do
6      a ← 0;
7      for s ← 1 to k_S do
8          j_s ← 1;
9      while a < ℓ̂_i do
10         for s ← 1 to k_S do
11             h_s ← first occurrence of T[i + a] in S_s[j_s..n] using D_s;
12             if h_s ≠ n_s + 1 then
13                 j_s ← h_s + 1;
14             else
15                 break while loop;
16         a ← a + 1;
17     max ← max{a, max};
18 return max
```

▶ **Theorem 18.** *Algorithm 2 solves the (k_S, k_T)-LCSS Problem (where $k_T \geq 1$) in $O(M + (N + m\ell k_S)\log|\Sigma|))$ time where $M = \sum_{i=1}^{k_T} m_i$, $N = \sum_{i=1}^{k_S} n_i$, $\ell = \mathsf{LCSS}(\mathcal{S}, \mathcal{T})$ and $m = \min\{m_s \mid s \in [|k_S|]\}$. The algorithm uses linear $O(N + M)$ space.*

Proof. Notice that at every outer-iteration of Line 5, the algorithm computes a valid length a which is a length of a CSS of $\mathcal{S}$ and $\mathcal{T}$. Therefore, the output of the algorithm is at most $\mathsf{LCSS}(\mathcal{S}, \mathcal{T})$. On the other hand, consider a string X that is a longest CSS of $\mathcal{S}$ and $\mathcal{T}$. By definition it must be a substring of all strings in $\mathcal{T}$, and in particular of the shortest string $T \in \mathcal{T}$ that the algorithm chooses at Line Line 2. Let i be an index of an occurrence of X in T. Following our discussion, at the ith outer-iteration of Line 5, the algorithm will find $|X| = \mathsf{LCSS}(\mathcal{S}, \mathcal{T})$. Thus, the algorithm outputs at least $\mathsf{LCSS}(\mathcal{S}, \mathcal{T})$. To conclude the correctness, the algorithm outputs exactly $\mathsf{LCSS}(\mathcal{S}, \mathcal{T})$.

We now focus on the complexity of the algorithm. The computation of the algorithm of Lemma 17 at Line 2 takes $O(M)$ time and space by Lemma 17. The initialization of each D_s at Line 3 takes $O(n_S \log|\Sigma|)$ time, summing up to $O(N \log|\Sigma|)$ time. Finally, there are exactly m iteration of the outer-loop of Line 5. In each such iteration, the algorithm first initializes all j_s pointers in $O(k_S)$ time at Line 7. Then, the iteration executes at most ℓ iterations of the while-loop of Line 9. Each iteration of the while-loop iterates over all strings in $\mathcal{S}$ (at Line 10) and for each one of them invoke a call for some D_s that takes $O(\log|\Sigma|)$. Thus, the total time of all outer-iterations is $O(m(k_S + \ell k_S \log|\Sigma|)) = O(m\ell k_S \log|\Sigma|)$. Summing up all factors, we obtain the stated running time. The space usage of the algorithm is linear in $N + M$ (which is the input size) due to Lemmas 2 and 17. ◀

5 LCSS Approximation

In this section we present a $(1-\varepsilon)$-approximation algorithm for the LCSS problem, for a given $\varepsilon > 0$. Let $\tilde{\ell} \in [m]$. We first describe how to compute a value x corresponding to the length of a common subsequence such that, if $\tilde{\ell} \leq \mathsf{LCSS}(S, T) < 2\tilde{\ell}$, then x is a $(1-\varepsilon)$-approximation of $\mathsf{LCSS}(S, T)$. We then repeat this procedure for $\log m$ exponentially increasing values of $\tilde{\ell}$, namely $\tilde{\ell} = 2^0, 2^1, \ldots, 2^{\lceil \log m \rceil}$, and return the maximum value of x obtained.

The algorithm is inspired by Algorithm 1, but instead of examining a common subsequence starting from every suffix of T, it only considers a subset of $O\left(\frac{m}{\varepsilon\tilde{\ell}}\right)$ suffixes. Specifically, define the set of positions in T: $P = \left\{1 + \lceil q\varepsilon\tilde{\ell} \rceil \;\middle|\; q \in \left[0, \left\lfloor \frac{m}{\varepsilon\tilde{\ell}} \right\rfloor\right]\right\}$. The algorithm iterates over suffixes of the form $T[i..m]$ for each $i \in P$. For each such suffix, it computes the length of the longest prefix of $T[i..i + 2\tilde{\ell} - 1]$ that appears as a subsequence of S.

Note that the algorithm does not process the entire suffix up to position m; instead, it restricts attention to a window of length $2\tilde{\ell}$.

In the following we show that this algorithm yields a $(1-\varepsilon)$ approximation of $\mathsf{LCSS}(S, T)$.

▶ **Lemma 19.** *Assume $\tilde{\ell} \leq \mathsf{LCSS}(S, T) < 2\tilde{\ell}$, the algorithm described above computes a $(1-\varepsilon)$ approximation of $\mathsf{LCSS}(S, T)$. Moreover, the algorithm runs (regardless of the assumption) in $O((m/\varepsilon + n)\log|\Sigma|)$ time.*

Proof. We first analyze the approximation ratio. Let $T[i..i + \mathsf{LCSS}(S, T) - 1]$ be a longest CSS. Let i' be the minimum $i' \in P$ in the set P such that $i' \geq i$. By the density of P we have that $i' \leq i + \varepsilon\tilde{\ell} \leq i + \varepsilon\mathsf{LCSS}(S, T)$ and therefore $i' - i \leq \varepsilon\mathsf{LCSS}(S, T)$. Moreover, since $T[i..i + \mathsf{LCSS}(S, T)]$ is a subsequence of S, we also have that $X = T[i'..i + \mathsf{LCSS}(S, T)]$ is a subsequence of S. By our assumption, when scanning the suffix $T[i'..i' + 2\tilde{\ell} - 1]$ the algorithm finds at least the length $x = |X|$. Thus, the reported value must be at least

$x \geq \mathsf{LCSS}(S,T) - (i' - i) \geq \mathsf{LCSS}(S,T) - \varepsilon\mathsf{LCSS}(S,T) = (1-\varepsilon)\mathsf{LCSS}(S,T)$. On the other hand since the algorithm always reports a length of a CSS, clearly the reported value is at most $\mathsf{LCSS}(S,T)$.

Finally, the algorithm executes $m/(\varepsilon\tilde{\ell})$ outer-iterations each one of them applies at most $2\tilde{\ell}$ inner iterations and each inner-iteration takes $O(\log|\Sigma|)$ time. Thus, the time complexity is $O((m/(\varepsilon\tilde{\ell}) \cdot 2\tilde{\ell} + n)\log|\Sigma|) = O((m/\varepsilon + n)\log|\Sigma|)$. ◀

By applying the algorithm $\log m$ times with exponential values of $\tilde{\ell} = 2^0, 2^1, \ldots 2^{\lceil \log m \rceil}$, it is clear that maximum reported value is a $(1-\varepsilon)$-approximation of $\mathsf{LCSS}(S,T)$, and that the running time is $O((m \log m/\varepsilon + n)\log|\Sigma|)$. Thus we obtain the following theorem.

▶ **Theorem 20.** *Let $\varepsilon > 0$. There exists an $O((\varepsilon^{-1} m \log m + n)\log|\Sigma|)$ time algorithm that returns a $(1-\varepsilon)$-approximation of $\mathsf{LCSS}(S,T)$.*

6 Conclusion and Future Work

In this paper, we study the time complexity of the LCSS problem. For a pair of strings we establish a quadratic lower bound and we introduce a near-linear time approximation algorithm. These results imply that requiring one of the common subsequence occurrences to be a substring does not reduce the quadratic time complexity of LCS. It seems that reducing the quadratic time requires restricting both input strings to consist of matching substrings, as done in the k-LCFg problem [17] of compiling a subsequence from at most k substrings in both input strings.

Our findings motivate the definition of additional LCS variants with substring constraints, to further explore the gap between the LCS and LCStr problems and to support the assumptions emerging from our findings.

References

1 Amir Abboud, Arturs Backurs, and Virginia Vassilevska Williams. Tight hardness results for LCS and other sequence similarity measures. In *2015 IEEE 56th Annual Symposium on Foundations of Computer Science*, pages 59–78, 2015. doi:10.1109/FOCS.2015.14.

2 Amir Abboud, Thomas Dueholm Hansen, Virginia Vassilevska Williams, and Ryan Williams. Simulating branching programs with edit distance and friends: or: a polylog shaved is a lower bound made. In *Proceedings of the Forty-Eighth Annual ACM Symposium on Theory of Computing*, STOC '16, pages 375–388, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2897518.2897653.

3 Amir Abboud and Aviad Rubinstein. Fast and Deterministic Constant Factor Approximation Algorithms for LCS Imply New Circuit Lower Bounds. In Anna R. Karlin, editor, *9th Innovations in Theoretical Computer Science Conference (ITCS 2018)*, volume 94 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 35:1–35:14, Dagstuhl, Germany, 2018. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ITCS.2018.35.

4 Amir Abboud, Ryan Williams, and Huacheng Yu. More applications of the polynomial method to algorithm design. In *Proceedings of the twenty-sixth annual ACM-SIAM symposium on Discrete algorithms*, pages 218–230. SIAM, 2014.

5 Shyan Akmal and Virginia Vassilevska Williams. Improved approximation for longest common subsequence over small alphabets. In Nikhil Bansal, Emanuela Merelli, and James Worrell, editors, *48th International Colloquium on Automata, Languages, and Programming, ICALP 2021, July 12-16, 2021, Glasgow, Scotland (Virtual Conference)*, volume 198 of *LIPIcs*, pages 13:1–13:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.ICALP.2021.13.

6 Mai Alzamel, Maxime Crochemore, Costas S. Iliopoulos, Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, Juliusz Straszynski, Tomasz Walen, and Wiktor Zuba. Quasi-linear-time algorithm for longest common circular factor. In Nadia Pisanti and Solon P. Pissis, editors, *30th Annual Symposium on Combinatorial Pattern Matching, CPM 2019, June 18-20, 2019, Pisa, Italy*, volume 128 of *LIPIcs*, pages 25:1–25:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPIcs.CPM.2019.25.

7 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.

8 Amihood Amir, Panagiotis Charalampopoulos, Costas S Iliopoulos, Solon P Pissis, and Jakub Radoszewski. Longest common factor after one edit operation. In *International Symposium on String Processing and Information Retrieval*, pages 14–26. Springer, 2017. doi:10.1007/978-3-319-67428-5_2.

9 Amihood Amir, Panagiotis Charalampopoulos, Solon P. Pissis, and Jakub Radoszewski. Longest common substring made fully dynamic. In Michael A. Bender, Ola Svensson, and Grzegorz Herman, editors, *27th Annual European Symposium on Algorithms, ESA 2019, September 9-11, 2019, Munich/Garching, Germany*, volume 144 of *LIPIcs*, pages 6:1–6:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPIcs.ESA.2019.6.

10 Amihood Amir, Zvi Gotthilf, and B. Riva Shalom. Weighted LCS. *J. Discrete Algorithms*, 8(3):273–281, 2010. doi:10.1016/J.JDA.2010.02.001.

11 Amihood Amir, Tzvika Hartman, Oren Kapah, B. Riva Shalom, and Dekel Tsur. Generalized LCS. *Theor. Comput. Sci.*, 409(3):438–449, 2008. doi:10.1016/J.TCS.2008.08.037.

12 Amihood Amir, Avivit Levy, Ely Porat, and B Riva Shalom. Dictionary matching with a few gaps. *Theoretical Computer Science*, 589:34–46, 2015. doi:10.1016/J.TCS.2015.04.011.

13 Hsing-Yen Ann, Chang-Biau Yang, and Chiou-Ting Tseng. Efficient polynomial-time algorithms for the constrained LCS problem with strings exclusion. *Journal of Combinatorial Optimization*, 28(4):800–813, 2014. doi:10.1007/S10878-012-9588-2.

14 Alberto Apostolico. Improving the worst-case performance of the Hunt-Szymanski strategy for the longest common subsequence of two strings. *Information Processing Letters*, 23(2):63–69, 1986. doi:10.1016/0020-0190(86)90044-X.

15 Alberto Apostolico and Concettina Guerra. The longest common subsequence problem revisited. *Algorithmica*, 2:315–336, 1987. doi:10.1007/BF01840365.

16 Sang Won Bae and Inbok Lee. On finding a longest common palindromic subsequence. *Theoretical Computer Science*, 710:29–34, 2018. doi:10.1016/J.TCS.2017.02.018.

17 Aranya Banerjee, Daniel Gibney, and Sharma V Thankachan. Longest common substring with gaps and related problems. In *32nd Annual European Symposium on Algorithms (ESA 2024)*, pages 16:1–16:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPIcs.ESA.2024.16.

18 David Becerra, Juan Mendivelso, and Yoan Pinzón. A multiobjective optimization algorithm for the weighted lcs. *Discrete Applied Mathematics*, 212:37–47, 2016. doi:10.1016/J.DAM.2015.11.005.

19 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.

20 Gary Benson, Avivit Levy, S. Maimoni, D. Noifeld, and B. Riva Shalom. Lcsk: A refined similarity measure. *Theor. Comput. Sci.*, 638:11–26, 2016. doi:10.1016/J.TCS.2015.11.026.

21 Lasse Bergroth, Harri Hakonen, and Timo Raita. A survey of longest common subsequence algorithms. In Pablo de la Fuente, editor, *Seventh International Symposium on String Processing and Information Retrieval, SPIRE 2000, A Coruña, Spain, September 27-29, 2000*, pages 39–48. IEEE Computer Society, 2000. doi:10.1109/SPIRE.2000.878178.

22 Guillaume Blin, Paola Bonizzoni, Riccardo Dondi, and Florian Sikora. On the parameterized complexity of the repetition free longest common subsequence problem. *Information Processing Letters*, 112(7):272–276, 2012. doi:10.1016/J.IPL.2011.12.009.

23 Karl Bringman and Marvin Künnemann. *Multivariate Fine-Grained Complexity of Longest Common Subsequence*, pages 1216–1235. Society for Industrial and Applied Mathematics, 2018. doi:10.1137/1.9781611975031.79.

24 Karl Bringmann, Vincent Cohen-Addad, and Debarati Das. A linear-time $n^{0.4}$-approximation for longest common subsequence. *ACM Trans. Algorithms*, 19(1):9:1–9:24, 2023. doi:10.1145/3568398.

25 Karl Bringmann and Marvin Künnemann. Quadratic conditional lower bounds for string problems and dynamic time warping. In *2015 IEEE 56th Annual Symposium on Foundations of Computer Science*, pages 79–97. IEEE, 2015. doi:10.1109/FOCS.2015.15.

26 Mauro Castelli, Riccardo Dondi, Giancarlo Mauri, and Italo Zoppis. Comparing incomplete sequences via longest common subsequence. *Theoretical Computer Science*, 796:272–285, 2019. doi:10.1016/J.TCS.2019.09.022.

27 Wun-Tat Chan, Yong Zhang, Stanley PY Fung, Deshi Ye, and Hong Zhu. Efficient algorithms for finding a longest common increasing subsequence. *Journal of Combinatorial Optimization*, 13(3):277–288, 2007. doi:10.1007/S10878-006-9031-7.

28 Panagiotis Charalampopoulos, Pawel 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.

29 Panagiotis Charalampopoulos, Tomasz Kociumaka, Solon P. Pissis, and Jakub Radoszewski. Faster algorithms for longest common substring. In Petra Mutzel, Rasmus Pagh, and Grzegorz Herman, editors, *29th Annual European Symposium on Algorithms, ESA 2021, September 6-8, 2021, Lisbon, Portugal (Virtual Conference)*, volume 204 of *LIPIcs*, pages 30:1–30:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.ESA.2021.30.

30 Debkumar Chowdhury, Kartik Sau, and Sanjukta Mishra. A new dna sequencing alignment methodology using the longest common subsequence technique. *American Journal of Applied Mathematics and Computing*, 1(3):5–9, 2020.

31 Maxime Crochemore, Costas S. Iliopoulos, Alessio Langiu, and Filippo Mignosi. The longest common substring problem. *Mathematical Structures in Computer Science*, 27(2):277–295, 2017. doi:10.1017/S0960129515000110.

32 Maxime Crochemore and Ely Porat. Fast computation of a longest increasing subsequence and application. *Information and Computation*, 208(9):1054–1059, 2010. doi:10.1016/j.ic.2010.04.003.

33 Lisa De Mattéo, Yan Holtz, Vincent Ranwez, and Sèverine Bérard. Efficient algorithms for longest common subsequence of two bucket orders to speed up pairwise genetic map comparison. *Plos one*, 13(12):e0208838, 2018.

34 Sebastian Deorowicz and Szymon Grabowski. Efficient algorithms for the longest common subsequence in k-length substrings. *Inf. Process. Lett.*, 114(11):634–638, 2014. doi:10.1016/J.IPL.2014.05.009.

35 Thomas Derrien, Jordi Estellé, Santiago Marco Sola, David G Knowles, Emanuele Raineri, Roderic Guigó, and Paolo Ribeca. Fast computation and applications of genome mappability. *PloS one*, 7(1):e30377, 2012.

36 Marko Djukanovic, Günther R Raidl, and Christian Blum. Anytime algorithms for the longest common palindromic subsequence problem. *Computers & Operations Research*, 114:104827, 2020. doi:10.1016/J.COR.2019.104827.

37 Lech Duraj, Marvin Künnemann, and Adam Polak. Tight conditional lower bounds for longest common increasing subsequence. *Algorithmica*, 81:3968–3992, 2019. doi:10.1007/S00453-018-0485-7.

38 David Eppstein, Zvi Galil, Raffaele Giancarlo, and Giuseppe F Italiano. Sparse dynamic programming i: linear cost functions. *Journal of the ACM (JACM)*, 39(3):519–545, 1992. doi:10.1145/146637.146650.

39 Martin Farach-Colton, Paolo Ferragina, and Shanmugavelayutham Muthukrishnan. On the sorting-complexity of suffix tree construction. *Journal of the ACM (JACM)*, 47(6):987–1011, 2000. doi:10.1145/355541.355547.

40 Effat Farhana and M Sohel Rahman. Doubly-constrained LCS and hybrid-constrained LCS problems revisited. *Information Processing Letters*, 112(13):562–565, 2012. doi:10.1016/J.IPL.2012.04.007.

41 Tomás Flouri, Emanuele Giaquinta, Kassian Kobert, and Esko Ukkonen. Longest common substrings with k mismatches. *Information Processing Letters*, 115(6-8):643–647, 2015. doi:10.1016/J.IPL.2015.03.006.

42 Darya Frolova, Helman Stern, and Sigal Berman. Most probable longest common subsequence for recognition of gesture character input. *IEEE Transactions on Cybernetics*, 43(3):871–880, 2013. doi:10.1109/TSMCB.2012.2217324.

43 Mitsuru Funakoshi, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Faster queries for longest substring palindrome after block edit. In *30th Annual Symposium on Combinatorial Pattern Matching (CPM 2019)*, pages 27:1–27:13. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPIcs.CPM.2019.27.

44 Shafi Goldwasser and Dhiraj Holden. The Complexity of Problems in P Given Correlated Instances. In Christos H. Papadimitriou, editor, *8th Innovations in Theoretical Computer Science Conference (ITCS 2017)*, volume 67 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 13:1–13:19, Dagstuhl, Germany, 2017. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ITCS.2017.13.

45 Anna Gorbenko and Vladimir Popov. The longest common parameterized subsequence problem. *Applied Mathematical Sciences*, 6(58):2851–2855, 2012.

46 Zvi Gotthilf, Danny Hermelin, and Moshe Lewenstein. Constrained lcs: hardness and approximation. In *Combinatorial Pattern Matching: 19th Annual Symposium, CPM 2008, Pisa, Italy, June 18-20, 2008 Proceedings 19*, pages 255–262. Springer, 2008. doi:10.1007/978-3-540-69068-9_24.

47 Zvi Gotthilf and Moshe Lewenstein. Approximating constrained lcs. In *International Symposium on String Processing and Information Retrieval*, pages 164–172. Springer, 2007. doi:10.1007/978-3-540-75530-2_15.

48 Dan Gusfield. *Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology*. Cambridge University Press, 1997. doi:10.1017/CBO9780511574931.

49 MohammadTaghi HajiAghayi, Masoud Seddighin, Saeedreza Seddighin, and Xiaorui Sun. Approximating longest common subsequence in linear time: Beating the n barrier. *SIAM Journal on Computing*, 51(4):1341–1367, 2022.

50 Daniel S Hirschberg. Algorithms for the longest common subsequence problem. *Journal of the ACM (JACM)*, 24(4):664–675, 1977. doi:10.1145/322033.322044.

51 Guo-Fong Huang, Chang-Biau Yang, Kuo-Tsung Tseng, and Kuo-Si Huang. Diagonal algorithms for the longest common subsequence problems with t-length and at least t-length substrings. In *Proceedings of the 37th Workshop on Combinatorial Mathematics and Computation Theory*, pages 119–127, 2020.

52 Lucas Hui. A practical algorithm to find longest common substring in linear time. *Computer Systems Science and Engineering*, 15, March 2000.

53 James W Hunt and Thomas G Szymanski. A fast algorithm for computing longest common subsequences. *Communications of the ACM*, 20(5):350–353, 1977.

54 Costas Iliopoulos, M Sohel Rahman, and Wojciech Rytter. Algorithms for two versions of LCS problem for indeterminate strings. *Journal of Combinatorial Mathematics and Combinatorial Computing*, 71:155, 2009.

55 Costas S Iliopoulos, Marcin Kubica, M Sohel Rahman, and Tomasz Waleń. Algorithms for computing the longest parameterized common subsequence. In *Combinatorial Pattern Matching: 18th Annual Symposium, CPM 2007, London, Canada, July 9-11, 2007. Proceedings 18*, pages 265–273. Springer, 2007.

56 Costas S Iliopoulos and M Sohel Rahman. Algorithms for computing variants of the longest common subsequence problem. *Theoretical Computer Science*, 395(2-3):255–267, 2008. doi:10.1016/J.TCS.2008.01.009.

57 Costas S Iliopoulos and M Sohel Rahman. New efficient algorithms for the LCS and constrained LCS problems. *Information Processing Letters*, 106(1):13–18, 2008. doi:10.1016/J.IPL.2007.09.008.

58 Costas S. Iliopoulos and M. Sohel Rahman. A new efficient algorithm for computing the longest common subsequence. *Theor. Comp. Sys.*, 45(2):355–371, June 2009. doi:10.1007/s00224-008-9101-6.

59 R. Impagliazzo and R. Paturi. On the complexity of k-SAT. *J. Comput. Syst. Sci.*, 62(2):367–375, 2001. doi:10.1006/JCSS.2000.1727.

60 R. Impagliazzo, R. Paturi, and F. Zane. Which problems have strongly exponential complexity? *J. Comput. Syst. Sci.*, 63(4):512–530, 2001. doi:10.1006/JCSS.2001.1774.

61 Shunsuke Inenaga and Heikki Hyyrö. A hardness result and new algorithm for the longest common palindromic subsequence problem. *Information Processing Letters*, 129:11–15, 2018. doi:10.1016/J.IPL.2017.08.006.

62 Robert W. Irving and Campbell B. Fraser. Two algorithms for the longest common subsequence of three (or more) strings. In Alberto Apostolico, Maxime Crochemore, Zvi Galil, and Udi Manber, editors, *Combinatorial Pattern Matching*, pages 214–229. Springer Berlin Heidelberg, 1992. doi:10.1007/3-540-56024-6_18.

63 Orgad Keller, Tsvi Kopelowitz, and Moshe Lewenstein. On the longest common parameterized subsequence. *Theoretical Computer Science*, 410(51):5347–5353, 2009. doi:10.1016/J.TCS.2009.09.011.

64 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.

65 Tomasz Kociumaka, Tatiana Starikovskaya, and Hjalte Wedel Vildhøj. Sublinear space algorithms for the longest common substring problem. In *European Symposium on Algorithms*, pages 605–617. Springer, 2014. doi:10.1007/978-3-662-44777-2_50.

66 Avivit Levy and B Riva Shalom. A comparative study of dictionary matching with gaps: limitations, techniques and challenges. *Algorithmica*, 84(3):590–638, 2022. doi:10.1007/S00453-021-00851-6.

67 Rao Li, Jyotishmoy Deka, and Kaushik Deka. An algorithm for the longest common subsequence and substring problem. *Journal of Mathematics and Informatics*, 25:77–81, 2023. doi:10.22457/jmi.v25a08231.

68 David Maier. The complexity of some problems on subsequences and supersequences. *J. ACM*, 25(2):322–336, April 1978. doi:10.1145/322063.322075.

69 Shay Mozes, Dekel Tsur, Oren Weimann, and Michal Ziv-Ukelson. Fast algorithms for computing tree lcs. *Theoretical Computer Science*, 410(43):4303–4314, 2009. String Algorithmics: Dedicated to Professor Maxime Crochemore on the occasion of his 60th birthday. doi:10.1016/j.tcs.2009.07.011.

70 Eugene W Myers. An O(ND) difference algorithm and its variations. *Algorithmica*, 1(1):251–266, 1986. doi:10.1007/BF01840446.

71 Narao Nakatsu, Yahiko Kambayashi, and Shuzo Yajima. A longest common subsequence algorithm suitable for similar text strings. *Acta Informatica*, 18:171–179, 1982. doi:10.1007/BF00264437.

72 Deena Nath, Jitendra Kurmi, and Vipin Rawat. A survey on longest common subsequence. *International Journal for Research in Applied Science and Engineering Technology (IJRASET)*, 6:4552–4556, 2018.

73 Negev Shekel Nosatzki. Approximating the longest common subsequence problem within a sub-polynomial factor in linear time. *CoRR*, abs/2112.08454, 2021. arXiv:2112.08454.

74 Aviad Rubinstein and Zhao Song. Reducing approximate longest common subsequence to approximate edit distance. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1591–1600. SIAM, 2020. doi:10.1137/1.9781611975994.98.

75 Tatiana Starikovskaya and Hjalte Wedel Vildhøj. Time-space trade-offs for the longest common substring problem. In *Combinatorial Pattern Matching: 24th Annual Symposium, CPM 2013, Bad Herrenalb, Germany, June 17-19, 2013. Proceedings 24*, pages 223–234. Springer, 2013. doi:10.1007/978-3-642-38905-4_22.

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

77 Robert A. Wagner and Michael J. Fischer. The string-to-string correction problem. *J. ACM*, 21:168–173, 1974. doi:10.1145/321796.321811.

78 Lusheng Wang and Binhai Zhu. Algorithms and hardness for the longest common subsequence of three strings and related problems. In Franco Maria Nardini, Nadia Pisanti, and Rossano Venturini, editors, *String Processing and Information Retrieval*, pages 367–380. Springer Nature Switzerland, 2023. doi:10.1007/978-3-031-43980-3_30.

79 Qingguo Wang, Mian Pan, Yi Shang, and Dmitry Korkin. A fast heuristic search algorithm for finding the longest common subsequence of multiple strings. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 24, pages 1287–1292, 2010. doi:10.1609/AAAI.V24I1.7493.

80 Y Wang. Longest common subsequence algorithms and applications in determining transposable genes. *arXiv preprint arXiv:2301.03827*, 2023.

81 Peter Weiner. Linear pattern matching algorithms. In *14th Annual Symposium on Switching and Automata Theory (swat 1973)*, pages 1–11. IEEE, 1973. doi:10.1109/SWAT.1973.13.

82 Ryan Williams. A new algorithm for optimal 2-constraint satisfaction and its implications. *Theoretical Computer Science*, 348(2-3):357–365, 2005. doi:10.1016/J.TCS.2005.09.023.

83 Sun Wu, Udi Manber, Gene Myers, and Webb Miller. An O(NP) sequence comparison algorithm. *Information Processing Letters*, 35(6):317–323, 1990. doi:10.1016/0020-0190(90)90035-V.

84 Daxin Zhu, Lei Wang, Tinran Wang, and Xiaodong Wang. A space efficient algorithm for the longest common subsequence in k-length substrings. *Theoretical Computer Science*, 687:79–92, 2017. doi:10.1016/J.TCS.2017.05.015.