

# Longest Common Subsequence with Gap Constraints

Duncan Adamson, Maria Kosche, Tore Koß, Florin Manea, and Stefan Siemer

Department of Computer Science, University of Göttingen, Göttingen, Germany

**Abstract.** We consider the longest common subsequence problem in the context of subsequences with gap constraints. In particular, following Day et al. 2022, we consider the setting when the distance (i. e., the gap) between two consecutive symbols of the subsequence has to be between a lower and an upper bound (which may depend on the position of those symbols in the subsequence or on the symbols bordering the gap) as well as the case where the entire subsequence is found in a bounded range (defined by a single upper bound), considered by Kosche et al. 2022. In all these cases, we present efficient algorithms for determining the length of the longest common constrained subsequence between two given strings.

## 1 Introduction

A *subsequence* of a string  $w = w[1]w[2] \dots w[n]$ , where  $w[i]$  is a symbol from a finite alphabet  $\Sigma$  for  $i \in \{1, \dots, n\}$ , is a string  $v = w[i_1]w[i_2] \dots w[i_k]$ , with  $k \leq n$  and  $1 \leq i_1 < i_2 \leq \dots < i_k \leq n$ . The positions  $i_1, i_2, \dots, i_k$  on which the symbols of  $v$  appear in  $w$  are said to define the embedding of  $v$  in  $w$ .

In general, the concept of subsequences appears and plays important roles in many different areas of theoretical computer science such as: formal languages and logics (e. g., in connection to piecewise testable languages [62,63,38,39,40], or in connection to subword-order and downward-closures [32,47,46,66]); combinatorics on words (e. g., in connection to binomial equivalence, binomial complexity, or to subword histories [58,23,50,49,60,55,59]); the design and complexity of algorithms. To this end, we mention some classical algorithmic problems such as the computation of *longest common subsequences* or of the *shortest common supersequences* [17,34,35,53,54,56,6,11], the testing of the Simon congruence of strings and the computation of the arch-factorisation and universality of words [33,29,64,65,19,7,20,22,30,44]; see also [45] for a survey of some combinatorial algorithmic problems related to subsequence matching. Moreover, these problems and some other closely related ones have recently regained interest in the context of fine-grained complexity (see [13,14,1,3]). Nevertheless, subsequences appear also in more applied settings: for modelling concurrency [57,61,15], in database theory (especially *event stream processing* [5,31,67]), in data mining [51,52], or in problems related to bioinformatics [12].

Most problems related to subsequences are usually considered in the setting where the embedding of subsequences in words are arbitrary. However, in [21], a novel setting is considered, based on the intuition that, in practical scenarios,

some properties with respect to the *gaps* that are induced by the embeddings can be inferred. As such, [21] introduces the notion of *subsequences with gap constraints*: these are strings  $v$  which can be embedded by some mapping  $e$  in a word  $w$  in such a way that the *gaps* of the embedding, i. e., the factors between the images of the mapping, satisfy certain properties. The main motivation of introducing and studying this model of subsequences in [21] comes from database theory [41,42], and the properties which have to be satisfied by the gaps are specified either in the form of length constraints (i. e., bounds on the length of the gap) or regular-language constraints. We refer the reader to [21] for a detailed presentation of subsequences with gap constraints and their motivations, as well as a discussion of the various related models. The main results of [21] are related to the complexity of the matching problem: given two strings  $w$  and  $v$ , decide if there is an embedding  $e$  of  $v$  as a subsequence of  $w$ , such that the gaps induced by  $e$  fulfil some given length and regular gap constraints. A series of other complexity results related to analysis problems for the set of subsequences of a word, which fulfil a given set of gap constraints, were obtained. The results of [21] are further extended in [43], where the authors consider *subsequences in bounded ranges*: these are strings  $v$  which can be embedded by some mapping  $e$  in a word  $w$  in such a way that the range in which all the symbols of  $v$  are embedded has length at most  $B$ , for some given integer  $B$ . This investigation was motivated in the context of sliding window algorithms [24,25,26,27,28], and the obtained results are again related to the complexity of matching and analysis problems.

One of the most studied algorithmic problem for subsequences is the problem of finding the length of the longest common subsequence of two strings (for short LCS), see, e. g., [17,34,35,53,54,56,6] or the survey [11]. In this problem, we are given two strings  $v$  and  $w$ , of length  $m$  and  $n$ , respectively, over an alphabet of size  $\sigma$ , and want to find the largest  $k$  for which there exists a string of length  $k$  which can be embedded as a subsequence in both  $v$  and  $w$ . The results on LCS are efficient algorithms (in most of the papers cited above) but also conditional complexity lower bounds [3,1,2]. In particular, there is a folklore algorithm solving LCS in  $O(N)$  time, for  $N = mn$ , and, interestingly, the existence of an algorithm whose complexity is  $O(N^{1-c})$ , for some  $c > 0$ , would refute the Strong Exponential Time Hypothesis (SETH), see [1].

*Our Contributions.* In this paper, we investigate the LCS problem in the context of subsequences with gap-length constraints, which seem to have a strong motivation and many application (see [21,43] and the references therein). Clearly, in the model considered by [21], the gap constraints depend on the length of the subsequence, while in LCS this is not known, and we actually need to compute this length. So, the model of [21] needs to be adapted to the setting of LCS. One way to do this is to consider that all the gaps of the common subsequence we search for are restricted by the same pair of lower and upper bounds; in this case, the length of the common subsequence plays no role anymore. We extend this initial idea significantly. On the one hand, we consider the case when there is a constant number of different length constraints which restrict the gaps (and they are given alongside the words). A further extension is the case when we

are given, alongside the input words, an arbitrarily long tuple of gap-length constraints: the  $i^{\text{th}}$  gap constraint in this tuple refers to the gap between the  $i^{\text{th}}$  and  $(i + 1)^{\text{th}}$  symbol of the common subsequence we try to find (and it plays some role only if that subsequence has length at least  $i + 1$ ); clearly, the longest common subsequence can be as long as the input words, so it has at most length  $\min\{m, n\}$ . We also consider the case when the gap constraints are given by the actual letters bounding the gap. All these extensions of LCS refer to models of constrained subsequences, where the constraints are local, i. e., they depend on the embedding of the symbols bounding the gap. Finally, we also consider LCS in the case of subsequences in bounded ranges, where the upper bound  $B$  on the size of the ranges in which we look for subsequences is given as input; in this case, we have a global constraint on the embedding of the subsequence.

After defining these variants of the LCS, which seem interesting and well motivated to us, we propose efficient algorithms for each of them. In most cases, these algorithms are non-trivial extensions of the standard dynamic programming algorithm solving LCS. A quick overview of our results for variants of LCS, when the gaps are local: if all gap constraints are identical or we have a constant number of different gap constraints (and the sequence of gap constraints fulfils some additional synchronization condition) we obtain algorithms running in  $O(N)$  time; if we have arbitrarily many different constraints, we obtain an algorithm running in  $O(Nk)$ , where  $k$  is the length of the longest common constrained subsequence of the input words; if, moreover, the sequence of constraints is increasing, then the problem can be solved in  $O(N \text{ polylog } N)$ ; if the constraints on the gaps are defined according to both letters bounding them, we obtain an algorithm running in  $O(\min\{N\sigma \log N, N\sigma^2\})$ ; if the constraints on the gaps are defined according only to the letter coming after (respectively, before) them, we obtain an algorithm running in  $O(\min\{N \log N, N\sigma\})$ . In the case of subsequences in bounded range, we show an algorithm which runs in  $O(NB^{o(1)})$  time for the respective extension of LCS (i. e., it runs in  $O(NB^d)$ , for some  $0 < d < 1$ ), as well as an  $\frac{1}{3}$ -approximation algorithm running in  $O(N)$  time.

*Related Work.* With respect to algorithms, the results of [37] cover the case when all gap constraints are identical. In particular, [37] considers a variant of LCS where the lengths of the gaps induced by the embeddings of the common subsequence in the two input strings are all constrained by the same lower and upper bounds and, additionally, there is an upper bound on the absolute value of the difference between the lengths of the  $i^{\text{th}}$  gap induced in  $w$  and the  $i^{\text{th}}$  gap induced in  $v$ , for all  $i$ . The authors of that paper propose a quadratic-time algorithm for this problem and then derive more efficient algorithms in some particular cases. To the best of our knowledge, the case of multiple gap-length constraints was not addressed so far in the literature. In [1], the authors consider LCS for subsequences in a bounded range, called there LOCAL-2-LCS, as an intermediate step in showing complexity lower bounds for LCS; they only mention the trivial  $O(NB^2)$  algorithm solving it. The results of [16] lead to an  $O(N^{1+o(1)})$  solution for this problem; our solution builds on that approach.

With respect to lower bounds, LCS is a particular case for all the problems we consider in our paper, as we can simply take all the length constraints to be trivial:  $(0, n)$  in the case of gap constraints or  $B = n$  in the case of subsequences in bounded ranges. Therefore, for each of our problems, the existence of an algorithm whose worst case complexity is  $O(N^{1-c})$ , with  $c > 0$ , would refute SETH. Thus, if  $\sigma \in O(1)$  (i. e., when the input is over alphabets of constant size), most of our algorithms solving LCS with gap-length constraints are optimal (unless SETH is false) up to polylog-factors; the exceptions are the two cases when we do not impose any monotonicity or synchronization condition on the tuple of gap-constraints. If  $\sigma$  is not constant, the previous claim also does not hold anymore for the case when the constraints on the gaps are defined according to both letters bounding them. In the case of LCS for subsequences in a bounded range, [1] shows quadratic lower bounds even for  $B$  being polylogarithmic in  $n$ . Thus, unless  $B \in O(\text{polylog } N)$ , there is a super-logarithmic mismatch between the upper bound provided by our algorithm and the existing lower bound.

It is natural to ask what happens when we have non-trivial constraints, such as, e.g., constraints of the form  $(a, b)$  with  $a, b \in O(1)$ . In [21], it is shown that deciding whether there exists an embedding of a string  $v$  as a subsequence of another string  $w$ , such that this embedding satisfies a sequence of  $|v|$  constraints of the form  $(a, b)$  with  $a, b \leq 6$ , cannot be done in  $O(N^{1-c})$  time, with  $c > 0$ , unless SETH is false; moreover, the respective reduction can be modified so that the embedding fulfils our synchronization property for the case of  $O(1)$  distinct gap constraints. The respective decision problem can also be solved by checking whether the longest common constrained subsequence of  $v$  and  $w$ , where the gap-length constraints for the common subsequence are exactly those defined for the embedding of  $v$  in  $w$ , has length  $|v|$ . So, the same lower bound from [21] (which coincides with the lower bound for the classical LCS problem) holds for the constrained LCS problem, even when we have a constant number of constant gap-length constraints, fulfilling, on top, the aforementioned synchronization property also. Due to page limitations some proofs are omitted in this version; see the full version [4].

## 2 Preliminaries

Let  $\mathbb{N} = \{1, 2, \dots\}$  be the set of natural numbers,  $[n] = \{1, \dots, n\}$ , and  $[m : n] = [n] \setminus [m - 1]$ , for  $m, n \in \mathbb{N}$ . For  $(a, b), (c, d) \in \mathbb{N}^2$ , we write  $(a, b) \subseteq (c, d)$  if and only if  $c \leq a$  and  $b \leq d$ . All logarithms used in this paper are in base 2.

For a finite alphabet  $\Sigma$ ,  $\Sigma^+$  denotes the set of non-empty words (or strings) over  $\Sigma$  and  $\Sigma^* = \Sigma^+ \cup \{\varepsilon\}$  (where  $\varepsilon$  is the empty word). For a word  $w \in \Sigma^*$ ,  $|w|$  denotes its length (in particular,  $|\varepsilon| = 0$ ). We set  $w^0 = \varepsilon$  and  $w^k = ww^{k-1}$  for every  $k \geq 1$ . For a word  $w$  of length  $n$  and some  $i \in [n]$ , we denote by  $w[i]$  the letter on the  $i^{\text{th}}$  position of  $w$ , so  $w = w[1]w[2] \dots w[n]$ . For every  $i, j \in [|w|]$ , we define  $w[i : j] = w[i]w[i+1] \dots w[j]$  if  $i \leq j$ , and  $w[i : j] = \varepsilon$ , if  $i > j$ . For  $w \in \Sigma^*$ , we define  $\text{alph}(w) = \{b \in \Sigma \mid b \text{ occurs at least once in } w\}$ . The strings  $w[i : j]$  are called *factors* of the string  $w$ ; if  $i = 1$  (respectively,  $j = |w|$ ), then  $w[i : j]$  is called a *prefix* (respectively, *suffix*) of  $w$ . For simplicity, for a word  $w$  and two natural numbers  $m \leq n$ , we write  $w \in [m, n]$  if  $m \leq |w| \leq n$ .

For an  $m \times n$  matrix  $M = (M[i, j])_{i \in [m], j \in [n]}$  and two sets  $I \subset [m], J \subset [n]$ , let  $M[I, J]$  be the submatrix  $(M[i, j])_{i \in I, j \in J}$  consisting in the elements which are at the intersection of row  $M[i, \cdot]$  and column  $M[\cdot, j]$  for  $i \in I$  and  $j \in J$ .

Further, we define the notions of subsequence and subsequence with gap-length constraints, following [21]. Our definitions are based on the notion of *embedding*. For a string  $w$ , of length  $n$ , and a natural number  $k \in [n]$ , an embedding is a function  $e: [k] \rightarrow [n]$  such that  $i < j$  implies  $e(i) < e(j)$  for all  $i, j \in [k]$ . We say  $e$  is a *matching* embedding if  $e(k) = n$ . For strings  $u, w \in \Sigma^*$  with  $|u| \leq |w|$ , an embedding  $e: [|u|] \rightarrow [|w|]$  is an *embedding of  $u$  into  $w$*  if  $u = w[e(1)]w[e(2)] \dots w[e(k)]$ , then  $u$  is called a *subsequence of  $w$* .

For an embedding  $e: [k] \rightarrow [|w|]$  and every  $j \in [k - 1]$ , the  $j^{\text{th}}$  *gap of  $w$  induced by  $e$*  is the string  $\text{gap}_e(w, j) = w[e(j) + 1 : e(j + 1) - 1]$ . A  $t$ -tuple of *gap-length constraints* is a  $t$ -tuple  $gc = (C_1, C_2, \dots, C_t)$  with  $C_i = (\ell_i, u_i)$  and  $0 \leq \ell_i \leq u_i \leq n$  for every  $i \in [t]$ . We set  $gc[i] = C_i$  for every  $i \in [t]$ , and  $gc[1 : i] = (C_1, C_2, \dots, C_i)$ . We say that an embedding  $e$  *satisfies a  $(k - 1)$ -tuple of gap-length constraints  $gc$  with respect to a string  $w$*  if it has the form  $e: [k] \rightarrow [|w|]$ , and, for every  $i \in [k - 1]$ ,  $\ell_i \leq |\text{gap}_e(w, i)| \leq u_i$  (that is,  $\text{gap}_e(w, i) \in C_i$ ).

If there is an embedding  $e$  of  $u$  into  $w$  satisfying the gap constraints  $gc$ , we denote this by  $u \preceq_{gc} w$ . For a  $(k - 1)$ -tuple  $gc$  of gap constraints, let  $\text{SubSeq}(gc, w)$  be the set of all subsequences of  $w$  induced by embeddings satisfying  $gc$ , i. e.,  $\text{SubSeq}(gc, w) = \{u \mid u \preceq_{gc} w\}$ . The elements of  $\text{SubSeq}(gc, w)$  are also called the *gc-subsequences of  $w$* . For more details see [21].

We are interested in defining and investigating the longest common subsequence problem (LCS for short) in the context of subsequences with gap constraints. However, in the framework introduced in [21], the gap constraints depend on (the length of) the subsequence, and this is not known for the LCS problem. As such, we propose a series of problems where we introduce variants of LCS accommodating gap-length constraints. In all our problems, we have two input strings  $v$  and  $w$ , with  $|v| = m$  and  $|w| = n$  and  $m \leq n$ , and these strings are over an alphabet  $\Sigma = \{1, 2, \dots, \sigma\}$ , with  $\sigma \leq m$ . For the rest of this paper, let  $N = mn$ . We also consider w.l.o.g. that, when the input contains gap-length constraints, every individual constraint  $C = (\ell, u)$  fulfils  $0 \leq \ell \leq u \leq n$ .

First, some additional definitions. Let  $v, w \in \Sigma^*$  be two words; a word  $s$  is a common subsequence of  $v$  and  $w$  if  $s$  is a subsequence of both  $v$  and  $w$ . Let  $gc$  be a  $(k - 1)$ -tuple of gap-length constraints. A word  $s$  of length  $k$  is a *common gc-subsequence of  $v$  and  $w$*  if both  $s \preceq_{gc} w$  and  $s \preceq_{gc} v$  hold. Let  $\text{ComSubSeq}(gc, v, w)$  denote  $\text{SubSeq}(gc, v) \cap \text{SubSeq}(gc, w)$ .

A  $(k - 1)$ -tuple  $gc$  of gap-length constraints is called *increasing* if  $gc[i] \subseteq gc[i + 1]$ , for all  $i \in [k - 2]$ . Let  $gc$  be an increasing  $(k - 1)$ -tuple of gap-length constraints and let  $i \in [k - 2]$ . Assume  $s$  is a  $gc[1 : i]$ -subsequence embedded in  $w[1 : i']$ , such that the last position of  $s$  is mapped to  $i'$ , and  $t$  is a  $gc[1 : j]$ -subsequence embedded in  $w[1 : i']$  as well, such that the last position of  $t$  is mapped to  $i'$ , and  $j > i$ . If there exists  $a \in \Sigma$  such that the embedding of  $s$  in  $w[1 : i']$  can be extended to an embedding of  $sa$  in  $w[1 : i'']$ , which satisfies

$gc[1 : i + 1]$ , for some  $i'' > i'$ , then the embedding of  $t$  in  $w[1 : i']$  can be extended as well to an embedding of  $ta$  in  $w[1 : i'']$  which satisfies  $gc[1 : i + 1]$ .

A  $(k - 1)$ -tuple  $gc$  of gap-length constraints is called *synchronized* when it satisfies the property that for all  $i, j \in [k - 1]$ , if  $gc[i] = gc[j]$  and  $i \leq j$  then  $gc[i + e] \subseteq gc[j + e]$  for all  $e \geq 0$  such that  $i + e \leq j + e \leq m - 1$ ; for example, the tuple  $((0, 5)(0, 1)(0, 2)(0, 3)(0, 1)(0, 5)(0, 3)(0, 4))$  is synchronized. Let  $gc$  be a synchronized  $(k - 1)$ -tuple of gap-length constraints and let  $i \in [k - 2]$ . Assume  $s$  is a  $gc[1 : i]$ -subsequence embedded in  $w[1 : i']$ , such that the last position of  $s$  is mapped to  $i'$ , and  $t$  is a  $gc[1 : j]$ -subsequence embedded in  $w[1 : i']$  as well, such that the last position of  $t$  is mapped to  $i'$ , and  $j > i$  and  $gc[i + 1] = gc[j + 1]$ . Now, if there exists a letter  $a$  such that the embedding of  $s$  in  $w[1 : i']$  can be extended to an embedding of  $sa$  in  $w[1 : i'']$  which satisfies  $gc[1 : i + 1]$ , for some  $i'' > i'$ , then the embedding of  $t$  in  $w[1 : i']$  can be extended as well to an embedding of  $ta$  in  $w[1 : i'']$  which satisfies  $gc[1 : i + 1]$ .

The Longest Common Subsequence Problem (LCS) is defined as follows.

*Problem 1 (LCS).* Given  $v, w$ , compute the largest  $k \in [m]$  such that there exists a common subsequence  $s$  of both  $v$  and  $w$  with  $|s| = k$ .

We now extend LCS to the case of subsequences with gap constraints (for a more detailed discussion on variants of this problem, see the full version [4]). Firstly we consider the case when the constraints are *local*, as in [21]: they concern only the gaps occurring between two consecutive symbols of the subsequence.

*Problem 2 (LCS-MC).* Given  $v, w \in \Sigma^*$  and an  $(m - 1)$ -tuple of gap-length constraints  $gc$ , compute the largest  $k \in \mathbb{N}$  such that there exists a common  $gc[1 : k - 1]$ -subsequence  $s$  of  $v$  and  $w$ , with  $|s| = k$ . That is, find the largest  $k$  for which  $ComSubSeq(gc[1 : k - 1], v, w)$  is non-empty.

Clearly, LCS is a particular case of LCS-MC, where  $gc = ((0, n), \dots, (0, n))$ .

In LCS-MC the input tuple gap-length constraints contains arbitrarily many constraints (therefore the acronym MC in the name of the problem), as many as the maximum amount of gaps that a common subsequence of  $v$  and  $w$  may have (that is,  $m - 1$ ). We also consider LCS-MC for increasing tuples of gap-length constraints  $gc$ ; this variant is called LCS-MC-INC.

We consider two special cases of Problem LCS-MC, where all these constraints are either identical (i. e., LCS with one constraint) or drawn from a set of constant size (i. e., LCS with  $O(1)$  constraints), which seem interesting to us.

*Problem 3 (LCS-1C).* Given  $v, w \in \Sigma^*$  and an  $(m - 1)$ -tuple of identical gap-length constraints  $gc = ((\ell, u), \dots, (\ell, u))$ , compute the largest  $k \in \mathbb{N}$  such that there exists a common  $gc[1 : k - 1]$ -subsequence  $s$  of  $v$  and  $w$ , with  $|s| = k$ .

*Problem 4 (LCS- $O(1)$ C).* Given  $v, w \in \Sigma^*$  and an  $(m - 1)$ -tuple of gap-length constraints  $gc = ((\ell_1, u_1), \dots, (\ell_{m-1}, u_{m-1}))$ , where  $|\{(\ell_i, u_i) \mid i \in [m - 1]\}|$  (the number of distinct constraints of  $gc$ ) is in  $O(1)$ , compute the largest  $k \in \mathbb{N}$  such that there exists a common  $gc[1 : k - 1]$ -subsequence  $s$  of  $v$  and  $w$ , with  $|s| = k$ .

The general results obtained for LCS-MC are improved for LCS-O(1)C, by considering the latter problem in the restricted setting of synchronized gap-length constraints only. The resulting problem is called LCS-O(1)C-SYNC.

In the problems introduced so far, the gap between two consecutive symbols in the searched subsequence depends on the positions of these symbols inside the respective subsequence (i.e., the gap between the  $i^{\text{th}}$  and  $i + 1^{\text{th}}$  symbols is always the same). That is, the actual symbols of the subsequence play no role in defining the constraints; it is only the length of the subsequence which is important. For the next problems, the constraints on a gap between consecutive symbols are determined by one or both symbols bounding the respective gap, and do not depend on the position of the gap inside the subsequence.

For this, we first need to modify our setting. Let  $left : \Sigma \rightarrow [n] \times [n]$  and  $right : \Sigma \rightarrow [n] \times [n]$  be two functions, defining the gap constraints. For an embedding  $e : [k] \rightarrow [n]$ , we say that  $e$  satisfies the gap constraints defined by  $(left, right)$  with respect to a string  $x$  if for every  $i \in [k - 1]$  we have that  $\text{gap}_e(x, i) \in left(x[e(i)]) \cap right(x[e(i + 1)])$ ; in other words,  $\text{gap}_e(x, i)$  has to simultaneously fulfil the constraints  $left(x[e(i)])$  and  $right(x[e(i + 1)])$ , defined by the symbols bounding that gap. If there is an embedding  $e$  of a string  $y$  into  $x$  satisfying the gap constraints  $(left, right)$ , we denote this by  $y \preceq_{left, right} x$  and call  $y$  a  $(left, right)$ -subsequence of  $x$ . In the following algorithmic problems, functions  $g : \Sigma \rightarrow [n] \times [n]$  are given as sequences of  $\sigma$  tuples  $(a, g(a))_{a \in \Sigma}$ .

*Problem 5 (LCS- $\Sigma$ ).* Given two words  $v, w \in \Sigma^*$  and two functions  $left : \Sigma \rightarrow [n] \times [n]$  and  $right : \Sigma \rightarrow [n] \times [n]$ , compute the largest number  $k \in \mathbb{N}$  such that there exists a common  $(left, right)$ -subsequence  $s$  of  $v$  and  $w$ , with  $|s| = k$ .

When  $left(a) = (0, n)$  for all  $a \in \Sigma$  (respectively,  $right(a) = (0, n)$  for all  $a \in \Sigma$ ), the gap constraints are defined only by the function  $right$  (respectively,  $left$ ), and the problem LCS- $\Sigma$  is denoted LCS- $\Sigma$ R (respectively, LCS- $\Sigma$ L).

In the problems introduced so far, the constraints were local (in the sense that they were defined by consecutive problems in the subsequence). In the last problem we introduce, we build on the works [1,43], and consider subsequences which occur inside factors of bounded length of the input words. In particular, for a given integer  $B$ , a word  $s$  is a  $B$ -subsequence of  $w$  if there exists a factor  $w[i + 1 : i + B]$  of  $w$  containing  $s$  as subsequence, and we look for the largest common  $B$ -subsequence of two input words. This problem was called Local-2-Longest Common Subsequence in [1], but as the constraint acts now globally on the subsequence, we prefer to call it LCS-BR (LCS in bounded range),

*Problem 6 (LCS-BR).* Given  $v, w \in \Sigma^*$  and  $B \in [n]$ , compute the largest  $k \in \mathbb{N}$  such that there exists a common  $B$ -subsequence  $s$  of  $v$  and  $w$ , with  $|s| = k$ .

We note that in all our definitions we are given a single tuple of gap-length constraints, meaning that the embeddings of the common subsequence of  $v$  and  $w$  should both fulfil the same constraints. Alternatively, we could have as input one tuple  $gc_v$  of gap-length constraints for  $v$  and one tuple  $gc_w$  for  $w$ , constraining the embeddings of the common subsequence in  $v$  and  $w$ , respectively. In this

settings, the embeddings would depend both on the subsequence and on the target word, not only on the subsequence, as in the model used currently in the paper.

Firstly, let us note that the model in which we have a single tuple of gap constraints seems more natural to us, as the gaps allowed in an embedding of a subsequence on a word seem to correspond rather to (or be determined by) properties of the subsequence, not to properties of the text in which it is embedded. For instance, in [21] as well as in the work on which that paper builds [41], the gaps are defined for the string which one wants to embed as a subsequence in a larger string.

Secondly, most of our results hold as such for the case when we are given as input two sets of gap constraints instead of a single one. The only results that do not hold in an identical form are those which rely on 2D RMQ data structures, namely the solutions for LCS- $\Sigma$ L/R running in  $O(N \log N)$  and the solution for LCS- $\Sigma$  running in  $O(N\sigma \log N)$ ; in all these cases we need to extend the 2D RMQ structure to allow queries on rectangular submatrices (instead of quadratic only), and this leads to an increase in the complexities by a  $\log n$ -factor.

We briefly discuss the *computational model* we use to describe our algorithms, solving efficiently the problems described in Section 2. This model is the standard unit-cost RAM with logarithmic word size: for an input of size  $L$ , each memory word can hold  $\log L$  bits. Arithmetic and bitwise operations with numbers in  $[1 : L]$  are, thus, assumed to take  $O(1)$  time. Moreover, the numbers we are given as inputs (describing, e. g., the gap constraints) are given in binary encoding. In all the problems, we assume that we are given two words  $w$  and  $v$ , with  $|w| = n$  and  $|v| = m$  (so the size of the input is  $L = n + m$ ), over an alphabet  $\Sigma = \{1, 2, \dots, \sigma\}$ , with  $2 \leq |\Sigma| = \sigma \leq m$ . That is, we assume that the processed words are sequences of integers (called letters or symbols), each fitting in  $O(1)$  memory words. This is a common assumption in string algorithms: the input alphabet is said to be *an integer alphabet*. Moreover, as the problems deal with common subsequences, we can assume without loss of generality that the alphabets of the two words are identical. For more details on this computational model see, e. g., [18].

### 3 LCS with local gap constraints

*An initial approach for LCS-MC.* For all variants of LCS where the constraints are local (i. e., they depend on the position of the gap in the subsequence, or on the letters bounding it), the sets of subsequences which are candidates for  $s$  can be, in the worst case, of exponential size in  $N$ . Therefore, computing the respective sets for both input words, their intersection, and then finding the longest string in this intersection would result in an exponential time algorithm. LCS can be, however, solved by a dynamic programming approach (considered folklore) in  $O(N)$  time. Similarly, LCS-MC (and its particular cases LCS-1C and LCS-O(1)C, as well as LCS- $\Sigma$ ) can also be solved by a dynamic programming

approach, running in polynomial time. We describe this general idea for LCS-MC only (as it can be easily adapted to all other problems). This idea reflects, to a certain extent, a less efficient implementation of the folklore algorithm for LCS.

For input strings  $v, w$  and constraints  $gc = (C_1, \dots, C_{m-1})$  we define, for each  $p \in [m]$ , a matrix  $M_p \in \{0, 1\}^{m \times n}$ , where  $M_p[i, j] = 1$  if and only if there exists a string  $s_p$  with  $|s_p| = p$  and matching embeddings  $e_v, e_w$ , respectively into  $v[1 : i]$  and  $w[1 : j]$ , satisfying  $gc[1 : p-1]$ . We compute  $M_1$  by setting  $M_1[i, j] = 1$  if and only if  $v[i] = w[j]$ . Then we compute  $M_p$  recursively by dynamic programming: let  $C_{p-1} = (\ell, u)$  and note that  $M_p[i, j] = 1$  if and only if  $v[i] = w[j]$  and there are positions  $i'$  with  $\ell \leq i - i' - 1 \leq u$  and  $j'$  with  $\ell \leq j - j' - 1 \leq u$  such that there is a string  $s_{p-1}$  of length  $p - 1$  with matching embeddings into  $v[1 : i']$  and  $w[1 : j']$ , respectively, satisfying  $gc[1 : p - 2]$ . That is  $M_p[i, j] = 1$  if and only if there is a 1 in the submatrix  $M_{p-1}[I, J]$  with  $I = [i - u - 1 : i - \ell - 1]$  and  $J = [j - u - 1 : j - \ell - 1]$ . In the end, the length  $k$  of the longest common subsequence of  $v$  and  $w$  satisfying  $gc$  equals the largest  $p$  such that  $M_p$  is not the 0-matrix. A naïve implementation of this approach runs in  $O(N^2k)$  time.

A more efficient implementation is given in the following.

$$\begin{pmatrix}
 M[1, 1] & * & * & * & * & * & M[1, j] & * & M[1, m] \\
 * & * & * & * & * & * & * & * & * \\
 * & * & M[i_u, j_u] & * & M[i_u, j_\ell] & * & * & * & * \\
 * & * & * & * & * & * & * & * & * \\
 * & * & M[i_\ell, j_u] & * & M[i_\ell, j_\ell] & * & * & * & * \\
 * & * & * & * & * & * & * & * & * \\
 \hline
 M[1, j] & * & * & * & * & * & M[i, j] & * & * \\
 * & * & * & * & * & * & * & * & * \\
 M[n, 1] & * & * & * & * & * & * & * & M[n, m]
 \end{pmatrix}$$

Fig. 1: The computation of  $M[i, j]$  in Lemma 1 with  $x_u = x - u - 1$  and  $x_\ell = x - \ell - 1$  for  $x \in \{i, j\}$ .

**Lemma 1.** *LCS-MC can be solved in  $O(Nk)$  time, where  $k$  is the largest number for which there exists a common  $gc[1 : k - 1]$ -subsequence  $s$  of  $v$  and  $w$ .*

*Proof.* As mentioned above, we can compute  $M_1$  in  $O(N)$  time. So, let  $2 \leq p \leq m$ ,  $C_{p-1} = (\ell, u)$ , and  $d = |C_{p-1}| = u - \ell + 1$ . We want to compute the elements of  $M_p$  and assume that  $M_{p-1}$  was already computed. For convenience we treat  $M_{p-1}[i, j] = 0$  when either  $i < 1$  or  $j < 1$ .

We use a pair of  $m \times n$  matrices  $A$  and  $B$ , where  $A[i, j]$  stores the sum of (or equivalently the amount of 1s in)  $d$  consecutive entries  $M_{p-1}[i, j - d + 1], \dots, M_{p-1}[i, j]$  in the rows of  $M_{p-1}$ . Then  $A[i, 1] = M_{p-1}[i, 1]$  and  $A[i, j] =$

$A[i, j - 1] - M_{p-1}[i, j - d] + M_{p-1}[i, j]$  for all  $i \in [m]$  and  $j \in [2 : n]$ . The entry  $B[i, j]$  stores the sum of all entries  $M_{p-1}[i', j']$  with  $0 \leq i - i' < d$  and  $0 \leq j - j' < d$ . Again, for convenience, we treat all entries  $A[i, j]$  as 0 if either  $i < 1$  or  $j < 1$ . Then we compute  $B[i, j]$  as follows. We set  $B[1, 1] = M_{p-1}[1, 1]$ ,  $B[1, j] = B[1, j - 1] - M_{p-1}[1, j - d] + M_{p-1}[1, j]$ , and  $B[i, j] = B[i - 1, j] - A[i - d, j] + A[i, j]$  for all  $i \in [2 : m]$  and  $j \in [2 : n]$ .

Since the computation of each entry in  $A$  or  $B$  takes  $O(1)$  time, we can compute the matrices  $A$  and  $B$  in time  $O(N)$ . Now  $M_p[i, j] = 1$  if and only if there is a 1 in the submatrix  $M_{p-1}[I, J]$ , which is true if  $B[i - \ell - 1, j - \ell - 1] > 0$ . Hence we can compute  $M_p[i, j]$  in constant time,  $M_p$  in  $O(N)$  time, and the sequence  $M_1, \dots, M_{k+1}$  in time  $O(Nk)$ .  $\square$

*An  $O(N \log^2 N)$  time algorithm for LCS-MC-INC.* We now consider LCS-MC-INC, a variant of LCS-MC where the tuple  $gc$  is increasing. We begin with a lemma describing a data structure, which is then used to solve LCS-MC-INC.

**Lemma 2.** *Given an  $m \times n$  matrix  $M$  with all elements initially equal to 0, we can maintain a data structure (two dimensional segment tree)  $\mathcal{T}$  for  $M$ , so that we can execute the following operations efficiently:*

- *update $_{\mathcal{T}}(i', i'', j', j'', x)$ : set  $M[i, j] = \max\{M[i, j], x\}$ , for all  $i \in [i' : i'']$  and  $j \in [j' : j'']$ ; here,  $x$  is a natural number. Time:  $O(\log n \log m)$ .*
- *query $_{\mathcal{T}}(i, j)$ : return  $M[i, j]$ . Time:  $O(\log m)$ .*

*Proof (Sketch).* The idea is to maintain a two dimensional (2D, for short) Segment Tree  $\mathcal{T}$  for  $M$  (see, for instance, [10,9] for details about segment trees). The 2D Segment Tree  $\mathcal{T}$  is defined for the matrix  $M$  as follows (see, e.g., [48,36], and also note that this is a relatively standard data structure in competitive programming).

- We define a segment tree  $T$  for the range  $[1 : n]$ .
- In each node  $\alpha$  of the segment tree  $T$  we have a segment tree  $T_\alpha$  for the range  $[1 : m]$ .
- In each node  $\beta$  of the tree  $T_\alpha$  we store an integer value  $val(\beta)$ , which is initially 0.

Note that the nodes of  $T$  correspond to sub-ranges  $[x : y]$  of the range  $[1 : n]$ . So, assume that we have a node  $\alpha$  which corresponds to the range  $[a, b]$ . Then the nodes of  $T_\alpha$  correspond canonically to the range  $[a : b]$  (as they depend on  $\alpha$ ) but also on a range  $[c : d]$  of  $[1 : m]$  (as they are segment trees for  $[1 : m]$ ). So, the nodes of  $T_\alpha$  correspond to submatrices  $M[c : d][a : b]$  of  $M$ .

Also there is a bijection between the leaves of the trees  $T_\alpha$ , where  $\alpha$  is a leaf of  $T$ , and the elements  $M[i, j]$ , with  $i \in [m]$  and  $j \in [n]$ . When constructing the structure  $\mathcal{T}$  as above, we can compute and store for each  $M[i, j]$  a pointer to the leaf corresponding to it.

Now, we explain how the operations are performed (without going into details w.r.t. the standard usage of segment trees and the results regarding them).

Consider first  $\text{update}_{\mathcal{T}}(i', i'', j', j'', x)$ . We first use the segment tree  $T_1$  to identify the nodes  $\alpha_1, \dots, \alpha_e$ , with  $e \leq \log n$ , which correspond to a partition of the interval  $[j' : j'']$ . The process of identifying these nodes is implemented in the standard way, and requires  $O(\log n)$  time. Then, for each of the trees  $T_\alpha$ , with  $\alpha \in \{\alpha_1, \dots, \alpha_e\}$ , we identify the nodes  $\beta_1^\alpha, \dots, \beta_{e_\alpha}^\alpha$ , with  $e_\alpha \leq \log m$ , which correspond to a partition of the interval  $[i' : i'']$ . Again, these can be identified in  $O(\log m)$  time. Now, the nodes  $\beta_1^\alpha, \dots, \beta_{e_\alpha}^\alpha$ , for  $\alpha \in \{\alpha_1, \dots, \alpha_e\}$ , correspond to a set of  $O(\log n \log m)$  submatrices which partition the submatrix  $M[i' : i''][j' : j'']$  (whose elements we need to update). Further, for  $\alpha \in \{\alpha_1, \dots, \alpha_e\}$  and for  $\beta \in \{\beta_1^\alpha, \dots, \beta_{e_\alpha}^\alpha\}$ , we set  $\text{val}(\beta) = \max\{\text{val}(\beta), x\}$ .

The time complexity of this update operation is  $O(\log n \log m)$ .

To retrieve the current value of  $M[i, j]$ , and answer query  $\mathcal{T}(i, j)$ , we proceed as follows. Note first that the actual entry  $M[i, j]$  might not have been changed. Therefore, we need to account for the updates we did on the trees. However, this is not complicated. We retrieve the leaf which corresponds to  $M[i, j]$  (say that this is a leaf in a node  $T_\alpha$ ). Then, we move up in the tree until we reach the root of this tree, and compute a value  $\text{ret}$ . Initially,  $\text{ret} = M[i, j]$ . Then, when we reach node  $\beta$  of  $T_\alpha$ , we update  $\text{ret} \leftarrow \max\{\text{ret}, \text{val}(\beta)\}$ . After processing the root of  $T_\alpha$  we stop, and return  $\text{ret}$  as the correct value of  $M[i, j]$ .

The time complexity of this query operation is  $O(\log m)$ .

The correctness of this approach follows immediately from the properties of segment trees.  $\square$

Based on this data structure, we can show the following Lemma.

**Lemma 3.** *LCS-MC-INC can be solved in  $O(N \log^2 N)$ .*

*Proof. Main idea.* Our algorithm computes, one by one, the elements of an  $m \times n$  matrix  $M$  (whose elements are initially set to 0). The approach is to define  $M[i, j]$ , for each pair of positions  $(i, j) \in [m] \times [n]$  such that  $v[i] = w[j]$ , to equal the length  $p$  of the longest string  $s_p$  which has matching embeddings into  $v[1 : i]$  and  $w[1 : j]$ , respectively, satisfying  $gc[1 : p - 1]$ . Because  $gc$  is increasing (for all  $i \in [m - 2]$ ,  $gc[i] \subseteq gc[i + 1]$ ),  $p$  can be determined as follows. It is enough to extend with the symbol  $a = v[i] = w[j]$  (mapped to position  $i$  of  $v$  and position  $j$  of  $w$ ) the longest subsequence  $s_{p'}$ , with  $|s_{p'}| = p'$ , such that the embeddings of  $s_{p'}$  in  $v$  and  $w$  end on positions  $i'$  and  $j'$ , respectively, where the gap between  $i'$  and  $i$  and the gap between  $j'$  and  $j$  fulfil the gap constraint  $gc[p' + 1]$ . Indeed, this is enough: as  $gc$  is increasing, this longest subsequence can be extended in exactly the same way as any other shorter subsequence with the same properties. Then, to obtain  $s_p$ , it is enough to set  $p = p' + 1$  and extend  $s_{p'}$  with the letter  $a$ , mapped to  $v[i]$  and  $w[j]$  in the two embeddings, respectively.

However, when considering the position  $(i, j)$ , we do not know the value of  $p'$ , and, as such, we do not know the range where we need to look for  $i'$  and  $j'$ . Therefore, we need to find a way around this.

*Dynamic programming approach.* We now show how to compute the elements of  $M$ . In the case of the dynamic programming algorithms solving LCS, the element  $M[i, j]$  of the matrix  $M$  is computed by looking at some elements  $M[i', j']$ ,

with  $i' \leq i, j' \leq j, (i, j) \neq (i', j')$ . By the arguments presented above, such an approach does not seem to work directly for LCS-MC-INC. However, if we know the value  $p$  of some entry  $M[i, j]$ , we can be sure that  $M[i'', j''] \geq p + 1$  for all  $i''$  and  $j''$  such that  $i + \ell + 1 \leq i'' \leq i + u + 1$  and  $j + \ell + 1 \leq j'' \leq j + u + 1$ , where  $gc[p + 1] = (\ell, u)$ ; we store this information. Moreover, if we know already all the values  $M[i, j]$ , with  $i \leq i', j \leq j', (i, j) \neq (i', j')$ , then we have already seen (and stored) all possible values for  $M[i', j']$  (or, in other words, all possible subsequences that we can extend in order to get  $M[i, j]$ ), so we simply set  $M[i', j']$  to the largest such possible value.

So, we compute the elements  $M[i, j]$  one by one, by traversing the elements of  $M$  for  $i$  from 1 to  $m$ , for  $j$  from 1 to  $n$ , and proceed as follows. When we reach an element  $M[i, j]$  in our traversal of  $M$ , we simply set it permanently to its current value. Then, if we set  $M[i, j]$  to some value  $p$ , and  $gc[p] = (\ell, u)$ , we update each element  $M[i', j']$  of submatrix  $M[I, J]$ , where  $I = [i + \ell + 1 : i + u + 1]$  and  $J = [j + \ell + 1 : j + u + 1]$ , to be  $M[i', j'] = \max\{M[i', j'], p + 1\}$ .

*The algorithm.* First we define the matrix  $M$ , and initialize all its entries with 0. Then, we build the data structure  $\mathcal{T}$  from Lemma 2 for  $M$ . In an initial step, we set all values  $M[1, j] = 1$ , where  $v[1] = w[j]$ , and  $M[i, 1] = 1$ , where  $v[i] = w[1]$ ; this is done by using update-operations on  $\mathcal{T}$  (to set the entry  $M[i, j] = x$ , for some  $x > 0$ , given that  $M[i, j]$  was equal to 0, it is enough to execute  $\text{update}(i, i, j, j, x)$ ). Further, we execute the following procedure.

- 1: for  $i = 2$  to  $m$  do
- 2:     for  $j = 2$  to  $n$  do
- 3:         Set  $M[i, j] = \text{query}_{\mathcal{T}}(i, j) = p$ ;  $M[i, j]$  remains equal to  $p$  permanently;
- 4:         For  $(\ell, u) = gc[p]$   $\text{update}_{\mathcal{T}}(i + \ell + 1, i + u + 1, j + \ell + 1, j + u + 1, p + 1)$ .

The solution to LCS-MC-INC is the maximum element of  $M$ .

*Conclusion.* The correctness of our algorithm follows from the arguments presented above. The time complexity of the algorithm is  $O(N \log^2 N)$ , as we need  $O(N \log^2 N)$  time for the preprocessing (setting up  $\mathcal{T}$  and doing the initial updates on it). Then the 4-step procedure described above takes also  $O(N \log^2 N)$  time, as in each iteration of the inner loop we perform as the most time consuming operation an update on  $\mathcal{T}$ . Our claim follows.  $\square$

Summing up, we have shown the following theorem regarding LCS-MC.

**Theorem 1.** *LCS-MC can be solved in  $O(Nk)$  time, where  $k$  is the largest number for which there exists a common  $gc[1 : k - 1]$ -subsequence  $s$  of  $v$  and  $w$ . LCS-MC-INC can be solved in  $O(N \log^2 N)$ .*

*$O(N)$  solutions for LCS-1C and LCS- $O(1)$ C.* While this problem was already solved in [37], we also briefly describe our solution for it. Our approach is based on the following data-structures lemma, which are also used to solve some of the other problems we discuss here.

**Lemma 4.** *Let  $\Psi : [m] \times [n] \rightarrow \{0, 1\}$  be a predefined function, such that  $\Psi(i, j)$  can be retrieved in  $O(1)$  time. Given an  $m \times n$  matrix  $M$ , with all elements*

initially equal to 0, and four positive integers  $\ell_1 \leq u_1, \ell_2 \leq u_2$ , we can maintain a data structure  $\mathcal{D}$  for  $M$ , so that the following process runs in  $O(N)$  time:

- 1: for  $i = 1$  to  $m$  do
- 2:     update  $\mathcal{D}$  (set up for processing line  $i$ );
- 3:     for  $j = 1$  to  $n$  do
- 4:         update  $\mathcal{D}$  (set up for computing  $M[i, j]$ );
- 5:         use  $\mathcal{D}$  to retrieve  $\mathbf{m}$ , the maximum of the submatrix  $M[I, J]$   
            where  $I = [i - u_1 : i - \ell_1]$  and  $J = [j - u_2 : j - \ell_2]$ ;  
             $\mathbf{m}$  is set to be 0 when  $I$  or  $J$  are empty.
- 6:         if  $\Psi(i, j) = 1$  then set  $M[i, j] = \mathbf{m} + 1$ .

*Proof. Preprocessing phase.* In the preprocessing, we define  $\mathcal{D}$ . This data structures contains a double ended queue (deque)  $Q_f$  for each column  $M[:, f]$  of the matrix  $M$ , with  $f \in [n]$ , as well as an array  $Max$  with  $n$  elements.

*Main idea.* In general we maintain the following invariant property for deque  $Q_f$ : the content of the deque  $Q_f$  is the list of matrix entries (in order, from the first element in  $Q_f$  to the last element of  $Q_f$ )  $M[i_{1,f}, f], \dots, M[i_{e_f,f}, f]$  of  $M[:, j]$  such that the following hold:

- $i_{1,f} < i_{2,f} < \dots < i_{e_f,f}$ ,
- $M[i_{1,f}, f] > \dots > M[i_{e_f,f}, f]$ , and
- $M[i_{g,f}, f] > M[h, f]$  for all  $i_{g-1,f} < h < i_{g,f}$ , for  $g \in [2 : e_f]$ .
- After executing step 4 of the process above for some values  $i$  and  $j$ , the queues  $Q_f$ , with  $f \in J$ , only contain elements of the submatrix  $M[I, J]$ .

*Implementation.* Initially, all deques in  $\mathcal{D}$  are empty.

The data structure  $\mathcal{D}$  is updated as follows (in step 2 of our process): for  $f$  from 1 to  $n$ , we remove from the deque  $Q_f$  the element  $M[i - 1 - d_1, f]$ , if that was contained in  $Q_f$ . As we execute this step for all values  $i$ , it is clear that when we execute step 2 for  $i = a$ , then the first element of  $Q_f$  is  $M[e, f]$  for some  $e \geq i - 1 - d_1$ . Moreover, once this step is completed, we recompute the array  $Max$  such that  $M[f]$  is the maximum between the first element of  $Q_f$  (i. e., the greatest element of  $Q_f$ ) and  $M[i - d_1 + \ell_1, f]$ . For the array  $Max$  we construct in  $O(n)$  time data structures allowing us to answer Range Maximum Queries in  $O(1)$  time (see [8]). At this point, we can retrieve in  $O(1)$  time the maximum element in any subarray  $Max[a : b]$ , with  $1 \leq a \leq b \leq n$ .

Further,  $\mathcal{D}$  is updated as follows (in step 4 of our process).

- When we execute step 3 of our algorithm for some values  $i$  and  $j$ , we insert  $M[i - d_1 + \ell_1, j - d_2 + \ell_2]$  in  $Q_{j-d_2-\ell_2}$ .
- The insertion of a value  $x$  in the deque  $Q_f$  for some  $f$  is handled as follows: we traverse  $Q_f$  from last element towards the first and remove all values smaller or equal to  $x$ . When we meet an element strictly greater than  $x$  or we have emptied  $Q_f$ , we store the value  $x$  as the last element of  $Q_f$ . This  $x$  is now the smallest element of  $Q_f$  (while the first element in  $Q_f$  is the greatest element in  $Q_f$ ).

Note that after the execution of this update step, the first element of the deque  $Q_f$  is exactly the element  $Max[f]$ , for all  $f \in J$ .

So, using the Range Maximum Query structures constructed in step 2, we can implement step 5 as simply querying to find the greatest element of  $Max$  over the range  $J$ . This is returned in  $O(1)$  time.

*Conclusion.* The correctness of this implementation follows from the explanations given above. To analyse the complexity of the algorithm, we note first that step 2 takes  $O(n)$  time (and is executed  $m$  times), steps 5 and 6 take  $O(1)$  time (and are executed  $mn$  times). To see how much time is spent in the execution of step 4 over the entire execution of the algorithm it is enough to note that each element of  $M$  is inserted once in one of the deques stored in  $\mathcal{D}$ , and the time spent in step 4 is proportional to the number of elements removed from these deques. So, the time spent overall in the execution of this step is upper bounded by the number of elements inserted in the deques. Thus, step 4 adds at most  $O(mn)$  time to the overall complexity of the algorithm. In total, this means that the respective procedure runs  $O(N)$  time, in the implementation described here.  $\square$

We can now show immediately the following result.

**Theorem 2.** *LCS-1C can be solved in  $O(N)$  time.*

*Proof.* Let  $(\ell, u)$  be the single gap-length constraint appearing in  $gc$ . We define the  $m \times n$  matrix  $M$ , where  $M[i, j] = p$  if and only if  $p$  is the greatest number for which there exists a subsequence  $s_p$ , with  $|s_p| = p$ , such that there are two matching embeddings  $e_v$  and  $e_w$  of  $s_p$  into  $v[1 : i]$  and  $w[1 : j]$ , respectively, both satisfying  $gc[1 : p-1]$ . We have that  $M[i, j] = p$  if and only if  $v[i] = w[j]$  and  $p-1$  is the greatest number for which there exist  $i'$  and  $j'$  with  $i-u-1 \leq i' \leq i-\ell-1$  and  $j-u-1 \leq j' \leq j-\ell-1$  and  $M[i', j'] = p-1$ . Hence, the entries of  $M$  can be computed using Lemma 4, for  $u_1 = u_2 = u+1$ ,  $\ell_1 = \ell_2 = \ell+1$  and  $\Psi(i, j) = 1$  if and only if  $v[i] = w[j]$ .  $\square$

This result can be extended to the case of LCS- $O(1)$ C-SYNC. It is, however, open if a similar result holds for the unrestricted problem LCS- $O(1)$ C.

**Theorem 3.** *LCS- $O(1)$ C-SYNC can be solved in  $O(N)$  time, where the constant hidden by the  $O$ -notation depends linearly on the number  $h$  of distinct gap-length constraints of  $gc$ .*

The result of Theorem 3 holds, in fact, for a larger family of constraints, namely constraints whose elements can be partitioned in  $h \in O(1)$  classes, such that, for all  $1 \leq i < j \leq k-1$ , if  $i, j$  are in the same class of the partition then  $gc[i] = gc[j]$  and  $gc[i+e] \subseteq gc[j+e]$  for all  $e \geq 0$  such that  $i+e \leq j+e \leq m-1$ .

*Proof. Preprocessing phase.* Assume  $(\ell'_1, u'_1), \dots, (\ell'_h, u'_h)$  is an enumeration of the distinct constraints from the set  $\{(\ell_i, u_i) \mid i \in [m-1]\}$ , where  $h$  is a constant and  $(\ell'_g, u'_g) \leq (\ell'_{g+1}, u'_{g+1})$  (w.r.t. canonical ordering of pairs of natural numbers). In  $O(m+n)$  time, we can radix-sort the list constraints of  $(\ell_i, u_i)$ , with

$i \in [m - 1]$ , and obtain the list  $(\ell'_1, u'_1), \dots, (\ell'_h, u'_h)$ , as well as an array  $label$  with  $m - 1$  elements, where  $label[i] = j$  if and only if  $(\ell_i, u_i) = (\ell'_j, u'_j)$ ; we say, for simplicity, that  $label[i]$  defines the gap  $(\ell'_{label[i]}, u'_{label[i]})$ . Now we move on to the description of the main algorithm.

*Main idea.* Our approach is to compute for each pair of positions  $(i, j) \in [m] \times [n]$ , such that  $v[i] = w[j] = a$ , and each  $r \in [h]$  the longest common subsequence  $s_p$  of  $v[1 : i]$  and  $w[1 : j]$ , with  $|s_p| = p$  and  $label(p) = r$ , which fulfils  $gc[1 : p - 1]$  and the last symbol of  $s_p$  is mapped to  $v[i]$  in the embedding of  $s_p$  in  $v[1 : i]$  and to  $w[j]$  by the embedding of  $s_p$  in  $w[1 : j]$ . This is obtained by extending a common subsequence  $s_{p-1}$  of  $v$  and  $w$ , with  $|s_{p-1}| = p - 1$ , which fulfils  $gc[1 : p - 2]$  and whose last symbol is mapped to position  $i'$  of  $v$  and to position  $j'$  of  $w$ , such that the gap between  $i'$  and  $i$  and the gap between  $j'$  and  $j$  fulfil the gap constraint defined by  $r' = label[p - 1]$ .

The main observation here is that, due to the synchronization property of  $gc$  (for all  $i, j \in [m - 1]$ , if  $gc[i] = gc[j]$  and  $i \leq j$  then  $gc[i + e] \subseteq gc[j + e]$  for all  $e \geq 0$ ),  $p$  can be determined as follows. For each  $r' \in [h]$ , it is enough to extend with a symbol  $a$  (mapped to position  $i$  of  $v$  and position  $j$  of  $w$ ) only the longest subsequence  $s_{m_{r'}}$  such that  $label(m_{r'}) = r'$  and the embeddings of  $s_{m_{r'}}$  in  $v$  and  $w$  end on positions  $i'_{r'}$  and  $j'_{r'}$ , respectively, where the gap between  $i'$  and  $i$  and the gap between  $j'$  and  $j$  fulfil the gap constraint defined by  $r'$ . Indeed, this is enough: due to the synchronization of  $gc$ , this longest subsequence can be extended in exactly the same way as any other shorter subsequence with the same properties. Then, to obtain  $s_p$ , it is enough to extend the longest of the subsequences  $s_{m_{r'}}$ , for  $r' \in [h]$ , such that  $label[m_{r'} + 1] = r$ . This naturally leads to a dynamic programming algorithm, which we describe in the following.

*Dynamic programming.* For each  $r \in [h]$ , we define an  $m \times n$  matrix  $M_r$ , where  $M_r[i, j] = p$  if and only if  $v[i] = w[j]$  and  $p$  is the greatest number for which  $r$  and there exists a  $gc[1 : p - 1]$ -subsequence  $s_p$  of  $v[1 : i]$  and  $w[1 : j]$  for which there are two embeddings  $e_v$  and  $e_w$  of  $s_p$  into  $v[1 : i]$  and  $w[1 : j]$ , respectively, with  $e_v(p) = i, e_w(p) = j$ .

According to the definition of  $M_r$  and the observations made above, we have that  $M_r[i, j] = p$  if and only if  $v[i] = w[j]$  and  $p - 1$  is the greatest number for which  $label[p] = r$  and there exist  $i'$  and  $j'$  with  $i - u'_{r'} - 1 \leq i' \leq i - \ell'_{r'} - 1$  and  $j - u'_{r'} - 1 \leq j' \leq j - \ell'_{r'} - 1$  such that  $M_{r'}[i', j'] = p - 1$ , for some  $r' \in [h]$ . That is,  $M_r[i, j] = p$  if there exists an embedding of subsequence of length  $p - 1$  which fulfils the gap constraints and which can be extended (while still fulfilling the constraints) to a subsequence of length  $p$ , whose last symbol is embedded in  $v[i]$  and  $w[j]$ , respectively, and, moreover, the label of the  $p^{th}$  gap (the one following the newly found  $p^{th}$  symbol) is  $r$ .

So, to compute  $M_r[i, j]$  we first have to compute for all  $r' \in [h]$  the maximum value  $m_{r'}$  of  $M_{r'}[I_{r'}, J_{r'}]$ , for  $I_{r'} = [i - u'_{r'} - 1 : i - \ell'_{r'} - 1]$  and  $J_{r'} = [j - u'_{r'} - 1 : j - \ell'_{r'} - 1]$ . Then, we simply set  $M_r[i, j] = 1 + \max_{r' \in [h], label[m_{r'} + 1] = r} m_{r'}$ .

Now, what remains to be explained is how to retrieve the maximum value  $p_{r'}$  of  $M_{r'}[I_{r'}, J_{r'}]$ , for  $I_{r'} = [i - u'_{r'} - 1 : i - \ell'_{r'} - 1]$  and  $J_{r'} = [j - u'_{r'} - 1 : j - \ell'_{r'} - 1]$ . For this we use Lemma 4, as shown below.

*The algorithm.* We initialize the  $m \times n$  matrices  $M_r$ , for  $r \in [h]$ , such that all their entries are 0, and define for each of them a data structure  $\mathcal{D}_r$  as in Lemma 4, with the four input numbers being  $\ell'_r + 1, u'_r + 1, \ell'_r + 1, u'_r + 1$ . Then, we adapt the procedure of Lemma 4 to compute these matrices as follows:

- 1: for  $i = 1$  to  $m$  do
- 2:     update  $\mathcal{D}_{r'}$ , for  $r' \in [h]$ ;
- 3:     for  $j = 1$  to  $n$  do
- 4:         update  $\mathcal{D}_{r'}$ , for  $r' \in [h]$ ;
- 5:         for  $r' \in [h]$ , use  $\mathcal{D}_{r'}$  to retrieve  $m_{r'}$ , the maximum of  $M_{r'}[I_{r'}, J_{r'}]$   
            where  $I_{r'} = [i - u'_{r'} - 1 : i - \ell'_{r'} - 1]$  and  $J_{r'} = [j - u'_{r'} - 1 : j - \ell'_{r'} - 1]$ ;  
             $m_{r'}$  is set to be 0 when  $I_{r'}$  or  $J_{r'}$  are empty.
- 6:         Compute set  $max_r = \max_{r' \in [h], label[m_{r'}+1]=r} m_{r'}$ , for all  $r \in [h]$ ;
- 7:         for  $r \in [h]$ , if  $v[i] = w[j]$  then set  $M_r[i, j] = 1 + max_r$ .

The result of LCS-O(1)-SYNC is given by the maximum value stored in one of the matrices  $M_r$  at the end of the computation.

*Conclusion.* The correctness of the algorithm follows from the explanations given above. Its time complexity is  $O(N)$ , based on the result of Lemma 4 and on the fact that  $h \in O(1)$ .  $\square$

First, we analyse the problem LCS- $\Sigma R$ . The input of this problem consists in two words  $v$  and  $w$  and one function  $right : \sigma \rightarrow [n] \times [n]$  with  $right(a) = (\ell_a, u_a)$  for all  $a \in \Sigma$  (that is, for the  $right$ -function we have  $right(a) = (0, n)$  for all  $a \in \Sigma$ ). The approach we use is based on Lemma 4, as in the solution to LCS-O(1)C-SYNC.

**Lemma 5.** *LCS- $\Sigma R$  can be solved in  $O(N\sigma)$  time.*

*Proof. Main idea.* Our approach is to compute for each pair of positions  $(i, j) \in [m] \times [n]$ , such that  $v[i] = w[j] = a$ , the longest ( $right$ )-subsequence  $s_p$  of both  $v$  and  $w$ , where  $|s_p| = p$  and the last symbol of  $s_p$  is mapped to  $v[i]$  in the embedding of  $s_p$  in  $v[1 : i]$  and to  $w[j]$  by the embedding of  $s_p$  in  $w[1 : j]$ . This can be obtained by simply extending with the symbol  $a = v[i] = w[j]$  the longest ( $right$ )-subsequence  $s_r$ , of length  $r$ , whose last symbol is mapped to position  $i'$  of  $v$  and to position  $j'$  of  $w$ , such that the gap between  $i'$  and  $i$  and the gap between  $j'$  and  $j$  fulfil the gap constraint defined by  $right(a)$ ; clearly,  $p$  is then defined as  $r + 1$ .

*Dynamic Programming.* We compute an  $m \times n$  matrix  $M$ , where  $M[i, j]$  is length of the longest ( $right$ )-subsequence  $s_p$  of both  $v$  and  $w$ , where  $|s_p| = p$  and the last symbol of  $s_p$  is mapped to  $v[i]$  in the embedding of  $s_p$  in  $v[1 : i]$  and to  $w[j]$  by the embedding of  $s_p$  in  $w[1 : j]$ ;  $M[i, j] = 0$  if and only if  $v[i] \neq w[j]$ . To compute  $M[i, j]$  we need to retrieve the largest entry  $M_a$  of the submatrix  $M[I_a, J_a]$ , where  $I_a = [i - u_a - 1 : i - \ell_a - 1]$  and  $J_a = [j - u_a - 1 : j - \ell_a - 1]$ , and set  $M[i, j] = M_a + 1$ . We can proceed as follows.

*The algorithm.* We initialize the  $m \times n$  matrices  $M$  such that all their entries are 0, and define for each  $a \in \Sigma$  a data structure  $\mathcal{D}_a$  as in Lemma 4, for the matrix  $M$  with the four input numbers being  $\ell_a + 1, u_a + 1, \ell_a + 1, u_a + 1$ , where  $right(a) = (\ell_a, u_a)$ . Then, we adapt the procedure of Lemma 4 to work as follows:

- 1: for  $i = 1$  to  $m$  do
- 2:     update  $\mathcal{D}_b$ , for  $b \in \Sigma$ ;
- 3:     for  $j = 1$  to  $n$  do
- 4:         update  $\mathcal{D}_b$ , for  $b \in' \Sigma$ ;
- 5:         if  $w[j] = v[i]$ , let  $a = w[j]$ ;
- 5:         use  $\mathcal{D}_a$  to retrieve  $M_a$ , the maximum of  $M[I_a, J_a]$ 
  - where  $I_a = [i - u_a - 1 : i - \ell_a - 1]$  and  $J_a = [j - u_a - 1 : j - \ell_a - 1]$ ;
  - $M_a$  is set to be 0 when  $I_a$  or  $J_a$  are empty.
- 6:         set  $M[i, j] = 1 + M_a$ .

*Conclusion.* The correctness of the algorithm follows from the explanations given above. Its time complexity is  $O(N\sigma)$ , based on the result of Lemma 4 and on the fact that we need to maintain  $\sigma$  data structures  $\mathcal{D}_a$ , for  $a \in \Sigma$ , and each of them can be maintained in overall time  $O(N)$ .  $\square$

We now present another algorithm for LCS- $\Sigma$ R, running in  $O(N \log m)$  time.

This algorithm uses a special case of the two-dimensional Range Maximum Query data structure (for short, RMQ), which is able to answer maximum queries on square sized submatrices of a matrix  $M$  of size  $m \times n$  in  $O(\log(m))$  time, and which allows a special type of updates needed in our solution for LCS- $\Sigma$ R. This structure extends the Sparse Table approach from [8].

*The two dimensional Range Maximum Query structure.* At its base, our RMQ data structure maintains an  $m \times n \times (1 + \lceil \log m \rceil)$  array  $RMQ[\cdot, \cdot, \cdot]$ , such that for  $1 \leq i \leq m$  and  $1 \leq j \leq n$  and  $0 \leq q \leq \lceil \log m \rceil$ ,  $RMQ[i, j, q]$  stores the maximum of the submatrix  $M[I, J]$  with  $I = [i - 2^q + 1 : i]$  and  $J = [j - 2^q + 1 : j]$ . For simplicity, we assume that  $M[i, j] = 0$  if  $i \leq 0$  or  $j \leq 0$ , and  $RMQ[i, j, q]$  is defined as 0 if  $i \leq 0$  or  $j \leq 0$ . Further, in a linear time preprocessing phase we can compute an array  $Q[1 : n]$  where, for  $h \in [m]$ , we have  $Q[h] = \max\{q \in \mathbb{N} \cup \{0\} \mid 2^q \leq h\}$  (in other words,  $Q[h] = \lfloor \log h \rfloor$ ). Let us now see how to retrieve the maximum of an arbitrary square submatrix of  $M[I, J]$  with  $I = [i' : i'']$  and  $J = [j' : j'']$  (with  $|i'' - i'| = |j'' - j'|$ ) in constant time, once we have computed the three dimensional array  $RMQ$ .

**Lemma 6.** *Given  $RMQ[\cdot, \cdot, \cdot]$ , we can retrieve  $\max M[I, J]$  in  $O(1)$ .*

*Proof.* As said, assume that we want to compute the largest value of  $M[I, J]$  with  $I = [i' : i'']$  and  $J = [j' : j'']$ . At first we need to determine the largest  $q$ , such that a square of size  $2^q$  completely fits into the square  $M[I, J]$ . That is to find a maximal  $q$  with  $2^q \leq |i' - i''| = |j' - j''| < 2^{q+1}$ , so  $q = Q[|i' - i''|]$ .

We now claim that the maximum of the values  $RMQ[i' + 2^q, j' + 2^q, q]$ ,  $RMQ[i' + 2^q, j'', q]$ ,  $RMQ[i'', j' + 2^q, q]$ ,  $RMQ[i'', j'', q]$  is a maximum in  $M[I, J]$ . We distinguish two cases.

**Case 1:**  $2^q = |i' - i''| = |j' - j''|$ . In this case we replace  $2^q$  in all four cases by  $|i' - i''|$  and get immediately four times the lookup  $RMQ[i'', j'', q]$ , that is by definition of  $RMQ[\cdot, \cdot, \cdot]$  the desired answer.

**Case 2:**  $2^q < |i' - i''| = |j' - j''| < 2^{q+1}$ . Let us have a look at the sets of values considered by the single  $RMQ$  look up operations. We have

$M[I_1, J_1], M[I_1, J_2], M[I_2, J_1], M[I_2, J_2]$  with  $I_1 = [i' : i' + 2^q - 1]$ ,  $J_1 = [j' : j' + 2^q - 1]$ ,  $I_2 = [i'' - 2^q + 1 : i'']$  and  $J_2 = [j'' - 2^q + 1 : j'']$ . Because we have  $2^q < |i' - i''| < 2^{q+1}$  and  $2^q < |j' - j''| < 2^{q+1}$ , we have  $I_1 \cap I_2 \neq \emptyset$  and  $J_1 \cap J_2 \neq \emptyset$ . Therefore, we completely cover all values of  $M[I, J]$ .  $\square$

Now that we have understood how the array  $RMQ$  is used, we see how to calculate  $RMQ[i, j, q]$  for all  $q$  initially. Clearly,  $RMQ[i, j, 0] = M[i, j]$ , for all  $i \in [m]$  and  $j \in [n]$ . Then we use a dynamic programming approach. We assume that all the entries  $RMQ[i, j, q - 1]$  for  $1 \leq i \leq m$  and  $1 \leq j \leq n$  have been computed already. Then,  $RMQ[i, j, q] = \max\{RMQ[i - 2^{q-1}, j - 2^{q-1}, q - 1], RMQ[i, j - 2^{q-1}, q - 1], RMQ[i - 2^{q-1}, j, q - 1], RMQ[i, j, q - 1]\}$ . Clearly, we need  $O(N)$  time to compute  $RMQ[\cdot, \cdot, q]$ . So, to compute the entire array  $RMQ$  we need  $O(N \log m)$  time.

*A RQM-based solution for LCS- $\Sigma R$ .* Our second solution to LCS- $\Sigma R$  is described in the following.

**Lemma 7.** *LCS- $\Sigma R$  can be solved in  $O(N \log m)$  time.*

*Proof. Main idea.* We compute an  $m \times n$  array  $M[\cdot, \cdot]$  such that  $M[i, j] = p$  if and only if  $p$  is the largest number for which there exists a (*right*)-subsequence  $s_p$  of  $v[1 : i]$  and  $w[1 : j]$  such that there are two embeddings  $e_v$  and  $e_w$  of  $s_p$  into  $v[1 : i]$  and  $w[1 : j]$ , respectively, with  $e_v(p) = i, e_w(p) = j$ .

The main observation is that  $M[i, j] = p$  if and only if  $v[i] = w[j] = a$ , where  $right(a) = (\ell_a, u_a)$ , and the maximum value in  $M[I, J]$ , for  $I = [i - u_a - 1 : i - \ell_a - 1]$  and  $J = [j - u_a - 1 : j - \ell_a - 1]$ , is  $p - 1$ . That is, there exist  $i' \in I$  and  $j' \in J$  such that  $M[i', j'] = p - 1$ , and this indicates the existence of a (*right*)-subsequence  $s_{p-1}$  of  $v[1 : i']$  and  $w[1 : j']$ , with  $|s_{p-1}| = p - 1$ , and whose last symbol is mapped, respectively, to  $v[i']$  and  $w[j']$ , and, moreover, this subsequence  $s_{p-1}$  can be extended to a (*right*)-subsequence  $s_p$  of  $v[1 : i]$  and  $w[1 : j]$  whose last symbol is mapped, respectively, to  $v[i]$  and  $w[j]$ .

*Preprocessing.* To begin with, we set all values of  $M$  to be 0. Then, for  $j$  from 1 to  $n$ , we set  $M[1, j] = 1$  if  $v[1] = w[j]$ , and for  $i$  from 1 to  $m$ , we set  $M[i, 1] = 1$  if  $v[i] = w[1]$ . We also compute the data structure  $RMQ[\cdot, \cdot, \cdot]$  for  $M$ .

*Dynamic programming algorithm.* We compute the entries of  $M[\cdot, \cdot]$  as follows:

- 1: for  $i = 2$  to  $m$  do
- 2:     for  $j = 2$  to  $n$  do
- 3:         get the constraint  $right(a) = (\ell_a, u_a)$  for  $a = v[i] = w[j]$ ;
- 4:         set  $I = [i - u_a - 1 : i - \ell_a - 1]$  and  $J = [j - u_a - 1 : j - \ell_a - 1]$ ;
- 5:         set  $M[i, j]$  as the maximum of  $M[I, J]$ , computed using  $RMQ$ ;
- 6:         set  $RMQ[i, j, 0] = M[i, j]$ ;
- 7:         for  $q = 1$  to  $\lceil \log m \rceil$
- 8:             Set  $RMQ[i, j, q] = \max\{RMQ[i - 2^{q-1}, j - 2^{q-1}, q - 1],$   
 $RMQ[i, j - 2^{q-1}, q - 1], RMQ[i - 2^{q-1}, j, q - 1], RMQ[i, j, q - 1]\}$

After we have computed all entries of  $M$ , we can simply compute its maximum in  $O(N)$  time, and return it as the output value for the LCS- $\Sigma R$ .

*Conclusion.* The correctness of our algorithm can be shown as follows. Clearly the entries of the arrays  $M[1, \cdot]$  and  $M[\cdot, 1]$  are correctly computed, and the data structure  $RMQ$  is correctly initialized. Assume now that  $M[i', j']$  is correctly computed and the data structure  $RMQ$  correctly returns  $RMQ[i', j', q]$  for all  $i' \leq i$  and  $j' \leq j$  such that  $(i', j') \neq (i, j)$ . By our observations made above, the computation from step 5 of  $M[i, j]$  is therefore correct. Moreover, once  $M[i, j]$  is computed, steps 6, 7, 8 ensure that our  $RMQ$  data structure correctly returns  $RMQ[i', j', q]$  for all  $i' \leq i$  and  $j' \leq j$  such that  $(i', j') \neq (i, j)$ . So, by induction, it follows that all the entries of  $M$  are correctly computed.

The overall time complexity of the algorithm is, clearly,  $O(N \log m)$ .  $\square$

*Solutions for LCS- $\Sigma L$  and LCS- $\Sigma$ .* Further, we consider the problem LCS- $\Sigma L$ , for which  $right(a) = (0, n)$  for all  $a \in \Sigma$ . The input of this problem consists in two words  $v, w$  and one function  $left$  with  $left(a) = (\ell_a, u_a)$  for all  $a \in \Sigma$ . We can immediately transform this problem into LCS- $\Sigma R$  with input words  $v^R$  and  $w^R$  (i. e., the mirror images of the input words) and the function  $right'$  which defines the gap constraints, where  $right'(a) = left(a)$ , for all  $a \in \Sigma$ . Then we can use the solutions we have seen in Lemmas 5 and 7 to get a solution for LCS- $\Sigma L$ .

**Theorem 4.** *LCS- $\Sigma R$ , LCS- $\Sigma L$  can be solved in  $O(\max\{N\sigma, N \log m\})$  time.*

Our approaches can be generalized to solve the general problem LCS- $\Sigma$ .

**Theorem 5.** *LCS- $\Sigma$  can be solved in  $O(\min\{N\sigma^2, N\sigma \log m\})$  time.*

*Proof. Main idea.* As in the case of LCS- $\Sigma R$ , our approach is to compute for each pair of positions  $(i, j) \in [m] \times [n]$ , such that  $v[i] = w[j] = a$ , the longest ( $left, right$ )-subsequence  $s_p$  of both  $v$  and  $w$ , where  $|s_p| = p$  and the last symbol of  $s_p$  is mapped to  $v[i]$  in the embedding of  $s_p$  in  $v[1 : i]$  and to  $w[j]$  by the embedding of  $s_p$  in  $w[1 : j]$ . This can be obtained by extending with the symbol  $a = v[i] = w[j]$  the longest ( $left, right$ )-subsequence  $s_r$ , of length  $r$ , whose last symbol, say  $b$ , is mapped to position  $i'$  of  $v$  and to position  $j'$  of  $w$ , such that the gap between  $i'$  and  $i$  and the gap between  $j'$  and  $j$  fulfils the gap constraint defined by the pair  $(left(b), right(a))$ ; clearly,  $p$  is then defined as  $r + 1$ .

To find, for some  $i$  and  $j$  such that  $v[i] = w[j]$ , the longest ( $left, right$ )-subsequence  $s_r$  which can be extended with the symbol  $a = v[i] = w[j]$  to a longer ( $left, right$ )-subsequence we proceed as follows. Let  $right(a) = (\ell_a, u_a)$ . For each letter  $b \in \Sigma$ , let  $left(b) = (\ell'_b, u'_b)$ . We compute the pair  $(\ell_{ab}, u_{ab}) = (\max\{\ell_a, \ell'_b\}, \min\{u_a, u'_b\})$  and let  $I_{ab} = [i - u_{ab} - 1 : i - \ell_{ab} - 1]$  and  $J_{ab} = [j - u_{ab} - 1 : j - \ell_{ab} - 1]$ . Then, we compute  $M_{ab}$  the maximum entry  $M[i'][j']$  of  $M[I_{ab}][J_{ab}]$  with  $v[i'] = w[j'] = b$ . Therefore, we need a mechanism allowing us to extract from  $M$  the maximum from a submatrix, but only consider the entries of this submatrix that correspond to a certain letter.

*Dynamic Programming.* To achieve this, we compute  $\sigma$   $m \times n$  matrices  $M_b$ , for all  $b \in \Sigma$  and an  $m \times n$  matrix  $M$ . We define  $M[i, j]$  as the longest (*left, right*)-subsequence  $s_p$  of both  $v$  and  $w$ , where  $|s_p| = p$  and the last symbol of  $s_p$  is mapped to  $v[i]$  in the embedding of  $s_p$  in  $v[1 : i]$  and to  $w[j]$  by the embedding of  $s_p$  in  $w[1 : j]$ ; clearly,  $M[i, j] \neq 0$  if and only if  $v[i] = w[j]$ . Then, for all  $b \in \Sigma$ ,  $M_b[i, j]$  is initialized as 0 and whenever we set  $M[i, j] = x$ , if  $v[i] = w[j] = b$  then we also set  $M_b[i, j] = x$ .

Now, to compute  $M[i, j]$ , if  $v[i] = w[j] = a$ , we need to retrieve, for all  $b \in \Sigma$ , the largest entry  $M_{ab}$  of the submatrix  $M_b[I_{ab}, J_{ab}]$ , and set  $M[i, j] = \max_{b \in \Sigma} M_{ab} + 1$ .

To avoid repeating the same algorithms again, we only describe how this is done informally. On the one hand, we can use the approach from Lemma 5, and maintain data structures  $\mathcal{D}_{ab}$  for each matrix  $M_b$ , for all  $a, b \in \Sigma$  (so, in total, maintain  $\sigma^2$  such data structures). Then, using the same algorithmic approach outlined in Lemma 4, the computation of all entries  $M[i, j]$  can be done in  $O(N\sigma^2)$  time. On the other hand, we can use the approach from Lemma 7, and maintain RMQ data structures for each matrix  $M_b$ , for all  $b \in \Sigma$ . Using the same algorithmic approach as in Lemma 7, the computation of all entries  $M[i, j]$  can be done in  $O(N\sigma \log m)$  time.

*Conclusion.* The correctness of this approach follows from the explanations given above and the discussions and proofs regarding LCS- $\Sigma$ R. Its time complexity is  $O(\min\{N\sigma^2, N\sigma \log m\})$ .  $\square$

## 4 LCS with Global Constraints

In this section, we present our solution to LCS-BR. First, we note the naïve solution: we consider every pair  $(v[i + 1 : i + B], w[j + 1 : j + B])$  of factors of length  $B$  of the two input words, respectively, and find their longest common subsequence, using the folklore dynamic programming algorithm for LCS. As each word of length  $n$  has  $n - B + 1$  factors of length  $B$ , this approach requires solving LCS for  $O((m - B)(n - B)) \subseteq O(N)$  words, with each such LCS-computation requiring  $O(B^2)$  time. This yields a total time complexity of  $O(NB^2)$ .

Here, we improve this by providing an  $O(NB^{o(1)})$  time algorithm via the *alignment oracles* provided by Charalampopoulos et al. [16]. Each such oracle is built for a pair of words  $v$  and  $w$ , with  $|v| = m$ ,  $|w| = n$  and  $N = mn$ , and is able to return the answer to queries asking for the length of the LCS between two factors  $v[i : i']$  and  $w[j : j']$ . One of the results of [16] is the following theorem.

**Theorem 6 ([16]).** *We can construct in  $N^{1+o(1)}$  time an alignment oracle for the words  $v$  and  $w$ , with  $\log^{2+o(1)} N$  query time.*

Theorem 6 can be used directly to build an alignment oracle  $\mathcal{A}$  for the input words  $w$  and  $v$  in  $O(N^{1+o(1)})$  time. Using this oracle, we can solve LCS-BR by making  $O(N)$  queries to  $\mathcal{A}$ , for every pair of indices  $i \leq m, j \leq n$ , each requiring  $O(\log^{2+o(1)} N)$  time. The total time complexity of this direct approach is therefore  $O(N^{1+o(1)})$ . We improve this approach by creating a set of smaller

oracles, allowing us to avoid the extra work required to answer LCS queries beyond the bounded range. For simplicity, assume w.l.o.g. that  $m$  and  $n$  are multiples of  $2B$ . Let, for all  $i$ ,  $w_i = w[iB + 1 : (i + 2)B]$ , or  $w_i = w[iB + 1 : n]$  if  $(i + 2)B > n$  and  $v_i = v[iB + 1 : (i + 2)B]$ , or  $v_i = v[iB + 1 : n]$  if  $(i + 2)B > m$ . Observe that every factor of length  $B$  of  $w$  appears in at least one subword  $w_i$  and every factor of length  $B$  of  $v$  appears in at least one subword  $v_i$ . Therefore, solving LCS-BR with input  $v_i, w_j$  for every  $i \in [0, m/B], j \in [0, n/B]$  would also give us the solution to LCS-BR for input words  $v, w$ .

To find the solution to LCS-BR with input  $v_i, w_j$ , we use Theorem 6 to construct an oracle  $\mathcal{A}_{i,j}$  for  $v_i$  and  $w_j$ . As  $|v_i| = |w_j| = 2B$ , the oracle  $\mathcal{A}_{i,j}$  can be constructed in  $O(B^{2+o(1)})$  time. The solution of LCS-BR for the input words  $v_i, w_j$  can be then determined by making  $O(B^2)$  queries to  $\mathcal{A}_{i,j}$ , each requiring  $O(\log^{2+o(1)} B)$  time. So, the total time complexity of solving LCS-BR for input words  $v_i, w_j$  is  $O(B^{2+o(1)})$ . As there are  $O(N/B^2)$  pairs of factors  $v_i, w_j$ , the time complexity of solving LCS-BR for words  $v, w$  is  $O(\frac{N}{B^2} B^{2+o(1)}) = O(NB^{o(1)})$ .

**Theorem 7.** *LCS-BR can be solved in  $O(NB^{o(1)})$  time.*

We complement our exact algorithm with an  $O(N)$  time constant-factor approximation algorithm. As before, we use the subwords  $v_i = v[iB + 1 : (i + 2)B]$  and  $w_j = w[jB + 1 : (j + 2)B]$ . Letting  $s$  be the longest common subsequence within a bounded range of length  $B$  between  $v$  and  $w$ , note that  $|s| \leq \max_{i,j \in [0, n/B]} LCS(v_i, w_j) \leq 3|s|$ . Therefore,  $\max_{i,j \in [0, n/B]} LCS(v_i, w_j)/3$  is at most a  $(1/3)$ -approximation of the longest common  $B$ -subsequence between  $v$  and  $w$ .

**Theorem 8.** *Given a pair of words  $v, w \in \Sigma^*$ , of length at most  $n$ , and let  $v_i = v[iB + 1 : (i + 2)B]$  and, respectively,  $w_j = w[jB + 1 : (j + 2)B]$ . The length of the longest common  $B$ -subsequence  $s$  between  $v$  and  $w$  has the following bound:*

$$\max_{i,j \in [n/B]} LCS(v_i, w_j)/3 \leq |s| \leq \max_{i,j \in [n/B]} LCS(v_i, w_j)$$

where  $LCS(v_i, w_j)$  is the length of longest common subsequence between the strings  $v_i$  and  $w_j$ . Further, these bounds can be found in  $O(N)$  time.

*Proof.* Let  $x, y \in [B + 1, n]$  be the pair of indices maximising  $LCS(v[x - B : x], w[y - B : y])$ , and let  $s$  be the longest common subsequence between  $v[x - B : x]$  and  $w[y - B : y]$ . Note that  $s$  must appear as a subsequence of  $v_i$  and  $w_j$  where  $i = \lfloor \frac{x}{B} \rfloor$  and  $j = \lfloor \frac{y}{B} \rfloor$ . Therefore, as  $LCS(v_i, w_j) \geq |s|$ , the length of  $s$  is at  $\max_{i,j \in [n/B]} LCS(v_i, w_j)$ .

On the other direction, assume for the sake of contradiction, that there exists some  $v_i, w_j$  such that  $|s| < LCS(v_i, w_j)/3$ . Let  $s'$  be the longest common subsequence between  $v_i$  and  $w_j$  such that  $|s'| > 3|s|$ . Now, let  $(x_1, y_1), (x_2, y_2), \dots, (x_{|s'|}, y_{|s'|})$  be a set of tuples of indices such that  $v[x_\ell] = w[y_\ell] = s'[\ell]$ . Observe that there exists a set of indices such that, for every  $\ell \in [2, |s'|]$ ,  $x - B \leq x_{\ell-1} < x_\ell \leq x$  and  $y - B \leq y_{\ell-1} < y_\ell \leq y$ . Therefore, if there exists some pair of indices  $x_\ell > iB + B, y_\ell \leq jB + B$ , there can not exist any

pair  $x_{\ell'} \leq iB+B, y_{\ell'} > jB+B$ . Conversely if there exists some pair  $x_{\ell} \leq iB+B, y_{\ell} > jB+B$ , then there can not exist any pair  $x_{\ell'} \leq iB+B, y_{\ell'} > jB+B$ . Therefore, there must be a set of sequences  $s'_1, s'_2, s'_3$  such that  $s' = s'_1, s'_2, s'_3$ , and:

- $s'_1$  is a subsequence of  $v_i[1 : B]$  and  $w_j[1 : B]$ .
- $s'_2$  is a subsequence of either  $v_i[1 : B]$  and  $w_j[B+1 : 2B]$  or  $v_i[B+1 : 2B]$  and  $w_j[1, B]$ .
- $s'_3$  is a subsequence of  $v_i[B+1 : 2B]$  and  $w_j[B+1 : 2B]$ .

As  $|s'_1| + |s'_2| + |s'_3| = |s'|$ , then the length of at least one of  $LCS(v_i[1 : B], w_j[1 : B])$ ,  $LCS(v_i[1 : B], w_j[B+1 : 2B])$ ,  $LCS(v_i[B+1 : 2B], w_j[1 : B])$  or  $LCS(v_i[B+1 : 2B], w_j[B+1 : 2B])$  must be at least  $|s'|/3 \leq |S|$ . Hence,

$$\max_{i,j \in [n/B]} LCS(v_i, w_j)/3 \leq |s| \leq \max_{i,j \in [n/B]} LCS(v_i, w_j).$$

The time complexity follows directly. □

## 5 Future Work

A series of problems remain open from our work. Can our results regarding LCS-MC be improved, at least in its particular case LCS-O(1)C? If not, can one show tight complexity lower bounds for these problems? Can the dependency of  $\Sigma$  from the solutions to LCS- $\Sigma$  and its variants be removed? We were not focused on shaving polylog factors from the time complexity of our algorithms, but it would be also interesting to see if this is achievable. Nevertheless, it would be interesting to address also the problem of efficiently computing the actual longest common constrained subsequences in the case of all addressed problems.

## References

1. Abboud, A., Backurs, A., Williams, V.V.: Tight hardness results for LCS and other sequence similarity measures. In: Guruswami, V. (ed.) IEEE 56th Annual Symposium on Foundations of Computer Science, FOCS 2015, Berkeley, CA, USA, 17-20 October, 2015. pp. 59–78. IEEE Computer Society (2015). <https://doi.org/10.1109/FOCS.2015.14>, <https://doi.org/10.1109/FOCS.2015.14>
2. Abboud, A., Rubinfeld, A.: Fast and deterministic constant factor approximation algorithms for LCS imply new circuit lower bounds. In: 9th Innovations in Theoretical Computer Science Conference, ITCS 2018, January 11-14, 2018, Cambridge, MA, USA. pp. 35:1–35:14 (2018). <https://doi.org/10.4230/LIPIcs.ITCS.2018.35>, <https://doi.org/10.4230/LIPIcs.ITCS.2018.35>
3. Abboud, A., Williams, V.V., Weimann, O.: Consequences of faster alignment of sequences. In: Automata, Languages, and Programming - 41st International Colloquium, ICALP 2014, Copenhagen, Denmark, July 8-11, 2014, Proceedings, Part I. pp. 39–51 (2014). [https://doi.org/10.1007/978-3-662-43948-7\\_4](https://doi.org/10.1007/978-3-662-43948-7_4)

4. Adamson, D., Kosche, M., Koß, T., Manea, F., Siemer, S.: Longest common subsequence with gap constraints. *CoRR to appear* (2023)
5. Artikis, A., Margara, A., Ugarte, M., Vansummeren, S., Weidlich, M.: Complex event recognition languages: Tutorial. In: Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems, DEBS 2017, Barcelona, Spain, June 19-23, 2017. pp. 7-10 (2017). <https://doi.org/10.1145/3093742.3095106>, <https://doi.org/10.1145/3093742.3095106>
6. Baeza-Yates, R.A.: Searching subsequences. *Theor. Comput. Sci.* **78**(2), 363-376 (1991)
7. Barker, L., Fleischmann, P., Harwardt, K., Manea, F., Nowotka, D.: Scattered factor-universality of words. In: Proc. DLT 2020. Lecture Notes in Computer Science, vol. 12086, pp. 14-28. Springer (2020)
8. Bender, M.A., Farach-Colton, M.: The LCA problem revisited. In: Gonnet, G.H., Panario, D., Viola, A. (eds.) LATIN 2000: Theoretical Informatics, 4th Latin American Symposium, Punta del Este, Uruguay, April 10-14, 2000, Proceedings. Lecture Notes in Computer Science, vol. 1776, pp. 88-94. Springer (2000). [https://doi.org/10.1007/10719839\\_9](https://doi.org/10.1007/10719839_9), [https://doi.org/10.1007/10719839\\_9](https://doi.org/10.1007/10719839_9)
9. de Berg, M., Cheong, O., van Kreveld, M.J., Overmars, M.H.: Computational geometry: algorithms and applications, 3rd Edition. Springer (2008), <https://www.worldcat.org/oclc/227584184>
10. de Berg, M., Cheong, O., van Kreveld, M.J., Overmars, M.H.: More Geometric Data Structures. Springer (2008). [https://doi.org/10.1007/978-3-540-77974-2\\_10](https://doi.org/10.1007/978-3-540-77974-2_10), <https://www.worldcat.org/oclc/227584184>
11. Bergroth, L., Hakonen, H., Raita, T.: A survey of longest common subsequence algorithms. In: de la Fuente, P. (ed.) Seventh International Symposium on String Processing and Information Retrieval, SPIRE 2000, A Coruña, Spain, September 27-29, 2000. pp. 39-48. IEEE Computer Society (2000). <https://doi.org/10.1109/SPIRE.2000.878178>, <https://doi.org/10.1109/SPIRE.2000.878178>
12. Bille, P., Gørtz, I.L., Vildhøj, H.W., Wind, D.K.: String matching with variable length gaps. *Theor. Comput. Sci.* **443**, 25-34 (2012). <https://doi.org/10.1016/j.tcs.2012.03.029>, <https://doi.org/10.1016/j.tcs.2012.03.029>
13. Bringmann, K., Chaudhury, B.R.: Sketching, streaming, and fine-grained complexity of (weighted) LCS. In: Proc. FSTTCS 2018. LIPIcs, vol. 122, pp. 40:1-40:16 (2018)
14. Bringmann, K., Künnemann, M.: Multivariate fine-grained complexity of longest common subsequence. In: Proc. SODA 2018. pp. 1216-1235 (2018)
15. Buss, S., Soltys, M.: Unshuffling a square is NP-hard. *J. Comput. Syst. Sci.* **80**(4), 766-776 (2014). <https://doi.org/10.1016/j.jcss.2013.11.002>, <https://doi.org/10.1016/j.jcss.2013.11.002>
16. Charalampopoulos, P., Gawrychowski, P., Mozes, S., Weimann, O.: An almost optimal edit distance oracle. In: Bansal, N., Merelli, E., Worrell, J. (eds.) 48th International Colloquium on Automata, Languages, and Programming, ICALP 2021, July 12-16, 2021, Glasgow, Scotland (Virtual Conference). LIPIcs, vol. 198, pp. 48:1-48:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2021). <https://doi.org/10.4230/LIPIcs.ICALP.2021.48>, <https://doi.org/10.4230/LIPIcs.ICALP.2021.48>

17. Chvátal, V., Sankoff, D.: Longest common subsequences of two random sequences. *Journal of Applied Probability* **12**(2), 306–315 (1975), <http://www.jstor.org/stable/3212444>
18. Crochemore, M., Hancart, C., Lecroq, T.: *Algorithms on strings*. Cambridge University Press (2007)
19. Crochemore, M., Melichar, B., Troníček, Z.: Directed acyclic subsequence graph — overview. *J. Discrete Algorithms* **1**(3-4), 255–280 (2003)
20. Day, J.D., Fleischmann, P., Kosche, M., Koř, T., Manea, F., Siemer, S.: The edit distance to  $k$ -subsequence universality. In: 38th International Symposium on Theoretical Aspects of Computer Science, STACS 2021, March 16-19, 2021, Saarbrücken, Germany (Virtual Conference). pp. 25:1–25:19 (2021). <https://doi.org/10.4230/LIPIcs.STACS.2021.25>, <https://doi.org/10.4230/LIPIcs.STACS.2021.25>
21. Day, J.D., Kosche, M., Manea, F., Schmid, M.L.: Subsequences with gap constraints: Complexity bounds for matching and analysis problems. In: Bae, S.W., Park, H. (eds.) 33rd International Symposium on Algorithms and Computation, ISAAC 2022, December 19-21, 2022, Seoul, Korea. *LIPIcs*, vol. 248, pp. 64:1–64:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2022). <https://doi.org/10.4230/LIPIcs.ISAAC.2022.64>, <https://doi.org/10.4230/LIPIcs.ISAAC.2022.64>
22. Fleischer, L., Kuffleitner, M.: Testing Simon’s congruence. In: *Proc. MFCS 2018*. *LIPIcs*, vol. 117, pp. 62:1–62:13 (2018)
23. Freydenberger, D.D., Gawrychowski, P., Karhumäki, J., Manea, F., Rytter, W.: Testing  $k$ -binomial equivalence. In: *Multidisciplinary Creativity*, a collection of papers dedicated to G. Păun 65th birthday. pp. 239–248 (2015), available in CoRR [abs/1509.00622](https://arxiv.org/abs/1509.00622)
24. Ganardi, M., Hucke, D., König, D., Lohrey, M., Mamouras, K.: Automata theory on sliding windows. In: *STACS*. *LIPIcs*, vol. 96, pp. 31:1–31:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2018)
25. Ganardi, M., Hucke, D., Lohrey, M.: Querying regular languages over sliding windows. In: *FSTTCS*. *LIPIcs*, vol. 65, pp. 18:1–18:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2016)
26. Ganardi, M., Hucke, D., Lohrey, M.: Randomized sliding window algorithms for regular languages. In: *ICALP*. *LIPIcs*, vol. 107, pp. 127:1–127:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2018)
27. Ganardi, M., Hucke, D., Lohrey, M.: Sliding window algorithms for regular languages. In: *LATA*. *Lecture Notes in Computer Science*, vol. 10792, pp. 26–35. Springer (2018)
28. Ganardi, M., Hucke, D., Lohrey, M., Starikovskaya, T.: Sliding window property testing for regular languages. In: *ISAAC*. *LIPIcs*, vol. 149, pp. 6:1–6:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2019)
29. Garel, E.: Minimal separators of two words. In: *Proc. CPM 1993*. *Lecture Notes in Computer Science*, vol. 684, pp. 35–53 (1993)
30. Gawrychowski, P., Kosche, M., Koř, T., Manea, F., Siemer, S.: Efficiently testing Simon’s congruence. In: 38th International Symposium on Theoretical Aspects of Computer Science, STACS 2021, March 16-19, 2021, Saarbrücken, Germany (Virtual Conference). pp. 34:1–34:18 (2021). <https://doi.org/10.4230/LIPIcs.STACS.2021.34>, <https://doi.org/10.4230/LIPIcs.STACS.2021.34>

31. Giatrakos, N., Alevizos, E., Artikis, A., Deligiannakis, A., Garofalakis, M.N.: Complex event recognition in the big data era: a survey. *VLDB J.* **29**(1), 313–352 (2020). <https://doi.org/10.1007/s00778-019-00557-w>, <https://doi.org/10.1007/s00778-019-00557-w>
32. Halfon, S., Schnoebelen, P., Zetzsche, G.: Decidability, complexity, and expressiveness of first-order logic over the subword ordering. In: *Proc. LICS 2017*. pp. 1–12 (2017)
33. Hebrard, J.J.: An algorithm for distinguishing efficiently bit-strings by their subsequences. *Theor. Comput. Sci.* **82**(1), 35–49 (22 May 1991)
34. Hirschberg, D.S.: Algorithms for the longest common subsequence problem. *J. ACM* **24**(4), 664–675 (1977). <https://doi.org/10.1145/322033.322044>, <https://doi.org/10.1145/322033.322044>
35. Hunt, J.W., Szymanski, T.G.: A fast algorithm for computing longest subsequences. *Commun. ACM* **20**(5), 350–353 (1977). <https://doi.org/10.1145/359581.359603>, <https://doi.org/10.1145/359581.359603>
36. Ibtihaz, N., Kaykobad, M., Rahman, M.S.: Multidimensional segment trees can do range updates in poly-logarithmic time. *Theor. Comput. Sci.* **854**, 30–43 (2021). <https://doi.org/10.1016/j.tcs.2020.11.034>, <https://doi.org/10.1016/j.tcs.2020.11.034>
37. Iliopoulos, C.S., Kubica, M., Rahman, M.S., Walen, T.: Algorithms for computing the longest parameterized common subsequence. In: Ma, B., Zhang, K. (eds.) *Combinatorial Pattern Matching, 18th Annual Symposium, CPM 2007, London, Canada, July 9–11, 2007, Proceedings. Lecture Notes in Computer Science*, vol. 4580, pp. 265–273. Springer (2007). [https://doi.org/10.1007/978-3-540-73437-6\\_27](https://doi.org/10.1007/978-3-540-73437-6_27), [https://doi.org/10.1007/978-3-540-73437-6\\_27](https://doi.org/10.1007/978-3-540-73437-6_27)
38. Karandikar, P., Kufleitner, M., Schnoebelen, P.: On the index of Simon’s congruence for piecewise testability. *Inf. Process. Lett.* **115**(4), 515–519 (2015)
39. Karandikar, P., Schnoebelen, P.: The height of piecewise-testable languages with applications in logical complexity. In: *Proc. CSL 2016. LIPIcs*, vol. 62, pp. 37:1–37:22 (2016)
40. Karandikar, P., Schnoebelen, P.: The height of piecewise-testable languages and the complexity of the logic of subwords. *Log. Methods Comput. Sci.* **15**(2) (2019)
41. Kleest-Meißner, S., Sattler, R., Schmid, M.L., Schweikardt, N., Weidlich, M.: Discovering event queries from traces: Laying foundations for subsequence-queries with wildcards and gap-size constraints. In: *25th International Conference on Database Theory, ICDT 2022. LIPIcs*, vol. 220, pp. 18:1–18:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2022). <https://doi.org/10.4230/LIPIcs.ICDT.2022.18>, <https://doi.org/10.4230/LIPIcs.ICDT.2022.18>
42. Kleest-Meißner, S., Sattler, R., Schmid, M.L., Schweikardt, N., Weidlich, M.: Discovering multi-dimensional subsequence queries from traces - from theory to practice. In: König-Ries, B., Scherzinger, S., Lehner, W., Vossen, G. (eds.) *Datenbanksysteme für Business, Technologie und Web (BTW 2023)*, 20. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme“ (DBIS), 06.–10. März 2023, Dresden, Germany, *Proceedings. LNI*, vol. P-331, pp. 511–533. Gesellschaft für Informatik e.V. (2023). <https://doi.org/10.18420/BTW2023-24>, <https://doi.org/10.18420/BTW2023-24>

43. Kosche, M., Koř, T., Manea, F., Pak, V.: Subsequences in bounded ranges: Matching and analysis problems. In: Lin, A.W., Zetsche, G., Potapov, I. (eds.) Reachability Problems - 16th International Conference, RP 2022, Kaiserslautern, Germany, October 17-21, 2022, Proceedings. Lecture Notes in Computer Science, vol. 13608, pp. 140–159. Springer (2022). [https://doi.org/10.1007/978-3-031-19135-0\\_10](https://doi.org/10.1007/978-3-031-19135-0_10), [https://doi.org/10.1007/978-3-031-19135-0\\_10](https://doi.org/10.1007/978-3-031-19135-0_10)
44. Kosche, M., Koř, T., Manea, F., Siemer, S.: Absent subsequences in words. In: Bell, P.C., Totzke, P., Potapov, I. (eds.) Reachability Problems - 15th International Conference, RP 2021, Liverpool, UK, October 25-27, 2021, Proceedings. Lecture Notes in Computer Science, vol. 13035, pp. 115–131. Springer (2021). [https://doi.org/10.1007/978-3-030-89716-1\\_8](https://doi.org/10.1007/978-3-030-89716-1_8), [https://doi.org/10.1007/978-3-030-89716-1\\_8](https://doi.org/10.1007/978-3-030-89716-1_8)
45. Kosche, M., Koř, T., Manea, F., Siemer, S.: Combinatorial algorithms for subsequence matching: A survey. In: Bordihn, H., Horváth, G., Vaszil, G. (eds.) Proceedings 12th International Workshop on Non-Classical Models of Automata and Applications, NCMA 2022, Debrecen, Hungary, August 26-27, 2022. EPTCS, vol. 367, pp. 11–27 (2022). <https://doi.org/10.4204/EPTCS.367.2>, <https://doi.org/10.4204/EPTCS.367.2>
46. Kuske, D.: The subtrace order and counting first-order logic. In: Proc. CSR 2020. Lecture Notes in Computer Science, vol. 12159, pp. 289–302 (2020)
47. Kuske, D., Zetsche, G.: Languages ordered by the subword order. In: Proc. FOS-SACS 2019. Lecture Notes in Computer Science, vol. 11425, pp. 348–364 (2019)
48. Lau, J., Ritossa, A.: Algorithms and hardness for multidimensional range updates and queries. In: Lee, J.R. (ed.) 12th Innovations in Theoretical Computer Science Conference, ITCS 2021, January 6-8, 2021, Virtual Conference. LIPIcs, vol. 185, pp. 35:1–35:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2021). <https://doi.org/10.4230/LIPIcs.ITCS.2021.35>, <https://doi.org/10.4230/LIPIcs.ITCS.2021.35>
49. Lejeune, M., Leroy, J., Rigo, M.: Computing the  $k$ -binomial complexity of the Thue-Morse word. In: Proc. DLT 2019. Lecture Notes in Computer Science, vol. 11647, pp. 278–291 (2019)
50. Leroy, J., Rigo, M., Stipulanti, M.: Generalized Pascal triangle for binomial coefficients of words. *Electron. J. Combin.* **24**(1.44), 36 pp. (2017)
51. Li, C., Wang, J.: Efficiently mining closed subsequences with gap constraints. In: SDM. pp. 313–322. SIAM (2008)
52. Li, C., Yang, Q., Wang, J., Li, M.: Efficient mining of gap-constrained subsequences and its various applications. *ACM Trans. Knowl. Discov. Data* **6**(1), 2:1–2:39 (2012)
53. Maier, D.: The complexity of some problems on subsequences and supersequences. *J. ACM* **25**(2), 322–336 (Apr 1978)
54. Masek, W.J., Paterson, M.: A faster algorithm computing string edit distances. *J. Comput. Syst. Sci.* **20**(1), 18–31 (1980). [https://doi.org/10.1016/0022-0000\(80\)90002-1](https://doi.org/10.1016/0022-0000(80)90002-1), [https://doi.org/10.1016/0022-0000\(80\)90002-1](https://doi.org/10.1016/0022-0000(80)90002-1)
55. Mateescu, A., Salomaa, A., Yu, S.: Subword histories and Parikh matrices. *J. Comput. Syst. Sci.* **68**(1), 1–21 (2004)
56. Nakatsu, N., Kambayashi, Y., Yajima, S.: A longest common subsequence algorithm suitable for similar text strings. *Acta Informatica* **18**, 171–179 (1982). <https://doi.org/10.1007/BF00264437>, <https://doi.org/10.1007/BF00264437>

57. Riddle, W.E.: An approach to software system modelling and analysis. *Comput. Lang.* **4**(1), 49–66 (1979). [https://doi.org/10.1016/0096-0551\(79\)90009-2](https://doi.org/10.1016/0096-0551(79)90009-2), [https://doi.org/10.1016/0096-0551\(79\)90009-2](https://doi.org/10.1016/0096-0551(79)90009-2)
58. Rigo, M., Salimov, P.: Another generalization of abelian equivalence: Binomial complexity of infinite words. *Theor. Comput. Sci.* **601**, 47–57 (2015)
59. Salomaa, A.: Connections between subwords and certain matrix mappings. *Theoret. Comput. Sci.* **340**(2), 188–203 (2005)
60. Seki, S.: Absoluteness of subword inequality is undecidable. *Theor. Comput. Sci.* **418**, 116–120 (2012)
61. Shaw, A.C.: Software descriptions with flow expressions. *IEEE Trans. Software Eng.* **4**(3), 242–254 (1978). <https://doi.org/10.1109/TSE.1978.231501>, <https://doi.org/10.1109/TSE.1978.231501>
62. Simon, I.: Hierarchies of events with dot-depth one — Ph.D. thesis. University of Waterloo (1972)
63. Simon, I.: Piecewise testable events. In: *Autom. Theor. Form. Lang.*, 2nd GI Conf. LNCS, vol. 33, pp. 214–222 (1975)
64. Simon, I.: Words distinguished by their subwords (extended abstract). In: *Proc. WORDS 2003*. TUCS General Publication, vol. 27, pp. 6–13 (2003)
65. Troníček, Z.: Common subsequence automaton. In: *Proc. CIAA 2002 (Revised Papers)*. Lecture Notes in Computer Science, vol. 2608, pp. 270–275 (2002)
66. Zetzsche, G.: The complexity of downward closure comparisons. In: *Proc. ICALP 2016*. LIPIcs, vol. 55, pp. 123:1–123:14 (2016)
67. Zhang, H., Diao, Y., Immerman, N.: On complexity and optimization of expensive queries in complex event processing. In: *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22–27, 2014*. pp. 217–228 (2014). <https://doi.org/10.1145/2588555.2593671>, <https://doi.org/10.1145/2588555.2593671>