The Edit Distance to k-Subsequence Universality

Joel D. Day
Loughborough University, UK

Pamela Fleischmann
Computer Science Department, Universität Kiel, Germany

Maria Kosche
Computer Science Department, Universität Göttingen, Germany

Tore Koß
Computer Science Department, Universität Göttingen, Germany

Florin Manea
Computer Science Department, Universität Göttingen, Germany
Campus-Institut Data Science, Göttingen, Germany

Stefan Siemer
Computer Science Department, Universität Göttingen, Germany

Abstract

A word u is a subsequence of another word w if u can be obtained from w by deleting some of its letters. In the early 1970s, Imre Simon defined the relation $\sim_k$ (called now Simon-Congruence) as follows: two words having exactly the same set of subsequences of length at most k are $\sim_k$-congruent. This relation was central in defining and analysing piecewise testable languages, but has found many applications in areas such as algorithmic learning theory, databases theory, or computational linguistics. Recently, it was shown that testing whether two words are $\sim_k$-congruent can be done in optimal linear time. Thus, it is a natural next step to ask, for two words w and u which are not $\sim_k$-equivalent, what is the minimal number of edit operations that we need to perform on w in order to obtain a word which is $\sim_k$-equivalent to u.

In this paper, we consider this problem in a setting which seems interesting: when u is a k-subsequence universal word. A word u with $\mathrm{alph}(u) = \Sigma$ is called k-subsequence universal if the set of subsequences of length k of u contains all possible words of length k over Σ. As such, our results are a series of efficient algorithms computing the edit distance from w to the language of k-subsequence universal words.

2012 ACM Subject Classification Theory of computation → Formal languages and automata theory; Theory of computation → Design and analysis of algorithms

Keywords and phrases Subsequence, Scattered factor, Subword, Universality, k-subsequence universality, Edit distance, Efficient algorithms

Digital Object Identifier 10.4230/LIPIcs.STACS.2021.25

Related Version *Full Version*: `https://arxiv.org/abs/2007.09192`

Funding The work of the four authors from Göttingen was supported by the DFG-grant 389613931.

1 Introduction

A word v is a subsequence (also called scattered factor or subword) of a word w if there exist (possibly empty) words $x_1, \ldots, x_{\ell+1}$ and $v_1, \ldots, v_\ell$ such that $v = v_1 \ldots v_\ell$ and $w = x_1 v_1 \ldots x_\ell v_\ell x_{\ell+1}$. That is, v is obtained from w by removing some of its letters.

The study of the relationship between words and their subsequences is a central topic in combinatorics on words and string algorithms, as well as in language and automata theory (see, e.g., the chapter *Subwords* by J. Sakarovitch and I. Simon in [55, Chapter 6] for an overview

38th International Symposium on Theoretical Aspects of Computer Science (STACS 2021).
Editors: Markus Bläser and Benjamin Monmege; Article No. 25; pp. 25:1–25:19
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

of the fundamental aspects of this topic). The concept of subsequences and its generalisations play an important role in various areas of theoretical computer science. For instance, in logic of automata theory, subsequences are used in the context of piecewise testability [60, 61], in particular to the height of piecewise testable languages [39, 40, 41], subword order [31, 44, 43], or downward closures [66]. In combinatorics on words, many concepts were developed around the idea of counting the occurrences of particular subsequences of a word, such as the k-binomial equivalence [54, 25, 46, 45], subword histories [59], and Parikh matrices [49, 56]. In the area of algorithms, subsequences appear, e.g., in classical problems such as the longest common subsequence [5, 10, 12], the shortest common supersequence [47], or the string-to-string correction [65]. From a practical point of view, subsequences are useful in scenarios related to bioinformatics, as well as in other areas where they model corrupted or lossy representations of an original string, see [57].

A major area of research related to subsequences is the study of the set of all subsequences of bounded length of a word, initiated by Simon in his PhD thesis [60]. In particular, Simon defined and studied (see [61, 55]) the relation $\sim_k$ (now called the Simon-Congruence) between words having exactly the same set of subsequences of length at most k, and used it in the study of piecewise testable languages, a class of regular languages with applications in learning theory, databases theory, or linguistics (see, e.g., [41] and the references therein). The surveys [51, 52] overview some of the extensions of Simon's seminal work from 1972 in various areas related to automata theory. Moreover, $\sim_k$ is a well-studied relation in the area of string algorithms, too. The problems of deciding whether two given words are $\sim_k$-equivalent, for a given k, and to find the largest k such that two given words are $\sim_k$-equivalent (and their applications) were heavily investigated in the literature, see, e.g., [34, 27, 62, 63, 18, 24] and the references therein. This year, optimal solutions were given for both these problems [6, 28]. In [6] it was shown how to compute the shortlex normal form of a given word in linear time, i.e., the minimum representative of a $\sim_k$-equivalence class w.r.t. shortlex ordering. This can be directly applied to test whether two words are $\sim_k$-equivalent: they need to have the same shortlex normal form. In [28], a data structure, called the Simon-Tree, was used to represent the equivalence classes induced by $\sim_k$ on the set of suffixes of a word, for all possible values of k, and then, given two words, a correspondence between their Simon-Trees was constructed to compute in linear time the largest k for which they are $\sim_k$-equivalent.

Motivation. As described above, asymptotically optimal algorithms are known for deciding whether two words w and u are $\sim_k$-equivalent. Thus, similarly to the case of other relations on strings (e.g., [4, 9]), it is natural to ask, for two words w and u, which are not $\sim_k$-equivalent, what is the minimal number of edit operations (edits, for short) that we need to perform on them in order to obtain two $\sim_k$-equivalent words. The edits we consider are the usual letter-insertion, -deletion, -substitution, and the scenario we assume is the following: we edit one word only (say w) and attempt to reach, with a minimal number of edits, an intermediate word which has the same subsequences of length k as the second input word (namely u).

This formulation is essentially an instance of a word-to-language edit distance problem, in which we wish to compute the distance between w and the language $L_{u,k}$ of words which are $\sim_k$-equivalent to u. It is well documented that word-to-language edit distance problems, alongside the classical word-to-word and also the language-to-language variants, are well motivated and have consequently been well studied (see, e.g., [64, 50, 33, 11, 38, 15, 16]). For instance, the authors of [13] write: *"the edit distance provides a quantitative measure of "how far apart" are two words, or a word from a language, or two languages, and it forms the basis for quantitatively comparing sequences, a problem that arises in many different* *areas, such as error-correcting codes, natural language processing, and computational biology; similarly, the edit distance between languages forms the foundations of a quantitative approach to verification".*

In our case, the languages $L_{u,k}$ are regular. In particular, for a given subsequence v of length k of u, we can easily construct a DFA recognising the language of all words containing v as a subsequence. Consequently, a finite automaton accepting $L_{u,k}$ can be obtained as a boolean combination of these DFAs. In fact, for a positive integer k, the set of all languages which can be written as the union of several languages $L_{u,k}$, where u are words of a finite set, is the class of k-piecewise testable languages [60, 61, 55], an important class of regular languages, with deep connections to logic and semigroup theory. Therefore, if we take as input the word w and the language $L_{u,k}$ given as an automaton $A_{u,k}$ with q states, we can solve our distance problem in time $O(|w|q^2)$ [64, 3]. However, this is not necessarily efficient, since even when $A_{u,k}$ is a minimal NFAs accepting $L_{u,k}$, the number q of states can be exponential in the size of the alphabet (and hence in the length of u; see [19]). Consequently, if we consider the input to be (w, u, k) rather than $(w, A_{u,k})$, the exact complexity remains unclear. We can, however, guarantee inclusion in NP as we can trivially rewrite w into the shortest word $u \in L_{u,k}$ using at most $|w| + |u|$ edits.

It is also worth pointing out that the order of w and u in the input matters: the number of edits necessarily applied to w order to reach a word w' such that $w' \sim_k u$ holds, is not generally equal to the number of edits needed to apply on u in order to reach a word u' such that $u' \sim_k w$. Consider, for example, the words $w = aba$, $u = aaabbbaaa$, and $k = 2$. We need one insertion to transform w into $abab$, which is $\sim_2$-equivalent to u, but we need two deletions to transform u into $aaabaaa$, which is $\sim_2$-equivalent to w. An intuitive explanation for this is that w is closer w.r.t. the edit distance to the set $L_{u,k}$ of words which are $\sim_k$-equivalent to u than u is to the set $L_{w,k}$ of words which are $\sim_k$-equivalent to w, and we only need to edit each of our words until it reaches the word which is closest to them from the respective sets.

Essentially, we are considering the word-to-language edit distance problem for regular languages (in fact, piecewise testable languages) which admit a particularly succinct representation: a single word u. One way to generalise this is to consider the edit distance from a word to the closure of a given language under $\sim_k$. The problem remains decidable when considering the closures of regular or context-free languages (the regular case can be solved in nondeterministic polynomial time when k is a constant, see [19]). On the other hand, we have already mentioned how taking the closure under $\sim_k$ can result in an exponential blow-up in the size of the representation of the language. Going in the other direction, one of the most natural restrictions is to consider only words u over an alphabet Σ for which all length-k subsequences over Σ occur, called *k-subsequence universal words* (called, for short, k-universal words) w.r.t. the alphabet Σ. This case is also among the ones for which the corresponding automata for $L_{u,k}$ may be exponentially large (see [19]), remaining thus non-trivial. This restriction forms the focus of our paper.

The focus of our paper. In some cases, the edit distance problem we introduced above admits an input-specification where the target language is defined in a way which is both easier-to-use and more succinct. One of these cases is the already mentioned language of k-subsequence universal words w.r.t. an alphabet $\Sigma = \{1, \dots, \sigma\}$. While this language can be defined by a word $(1 \cdot 2 \cdots \sigma)^k$ of length $k\sigma$ or by an NFA with $\Theta(2^\sigma)$ states, it can also be simply specified by the number k and the alphabet Σ (or even only the size of this alphabet).

The main contribution of our paper, described below, is the study of the following problem: given a word w and a number k, compute the minimum number of edits we need to apply to w in order to obtain a k-universal word w.r.t. $\mathrm{alph}(w)$ (see [19] for a discussion on why the alphabet Σ used in the definition of universality is chosen here to be the set $\mathrm{alph}(w)$ of letters occurring in the input word w). As such, we are interested in the edit distance from the input word w to the set of k-universal words w.r.t. $\mathrm{alph}(w)$. We give a series of efficient algorithms showing how to solve this problem.

This investigation seems interesting to us as, on the one hand, the language of k-universal words plays an interesting role in the picture described in the **Motivation** section above. On the other hand, the class of languages of k-universal words occurs prominently in the study of the combinatorial and language theoretic properties of subsequences and piecewise testable languages. Indeed, in [39, 40, 41] the authors define and use the notion of k-rich words in relation to the study of the height of piecewise testable languages. The class of k-rich words coincides with that of k-subsequence universal words, which were further investigated, from a combinatorial point of view, in [20, 6]. Moreover, the idea of universality is quite important in formal languages and automata theory. The classical universality problem (see, e.g., [36]) is whether a given language L (over an alphabet Σ, specified by an automaton or grammar) is equal to Σ^*. The works [53, 42, 29] and the references therein discuss many variants of and results on the universality problem for various language generating and accepting formalisms. The universality problem was considered for words [48, 21] and partial words [14, 30] w.r.t. their factors. More precisely, one is interested in finding, for a given ℓ, a word w over an alphabet Σ, such that each word of length ℓ over Σ occurs exactly once as a contiguous factor of w. De Bruijn sequences [21] fulfil this property and have many applications in computer science or combinatorics, see [14, 30] and the references therein. It is worth noting that in the case of factor-universality it makes sense to ask for words where each factor occurs exactly once, but in the case of subsequence universality this is a trivial restriction, as in each long-enough word there will be subsequences occurring more than once [6].

As such, investigating k-subsequence universality from an algorithmic perspective is motivated by, fits in, and even enriches this well-developed and classical line of research.

Our results. The maximum k for which a word w is k-universal is called *the universality index* of w, and denoted $\iota(w)$. Firstly, we note that when we want to increase the universality index of a word by edits, it is enough to use only insertions. Similarly, when we want to decrease the universality index of a word, it is enough to consider deletions. So, to measure the edit distance to the class of k-subsequence universal words, for a given k, it is enough to consider either insertions or deletions. However, changing the universality of a word by substitutions (both increasing and decreasing it) is interesting in itself as one can see the minimal number of substitutions needed to transform a word w into a k-universal word as the *Hamming distance* [32] between w and the set of k-universal words. Thus, we consider all these operations independently and propose efficient algorithms computing the minimal number of insertions, deletions, and substitutions, respectively, needed to apply to a given word w in order to reach the class of k-universal words (w.r.t. the alphabet of w), for a given k. The time needed to compute these numbers is $O(nk)$ in the case of deletions and substitutions, as well as in the case of insertions when $k \leq n$ (for larger values of k it is just the time complexity of computing $k\sigma - n$, which is the value of the distance in that case). These algorithms are presented in the Section 4, and work in optimal linear time for constant k.

These algorithms are based, like most edit distance algorithms, on a dynamic programming approach. However, implementing such an approach within the time complexities stated above does not seem to follow directly from the known results on the word-to-word or word-to-language edit distance. In particular, we do not explicitly construct any k-universal word nor any representation (e.g., automaton or grammar) of the set of k-universal words, when computing the distance from the input word w to this set. Rather, we obtain the k-universal word which is closest w.r.t. edit distance to w as a byproduct of our algorithms. In our approach, we first develop (Section 3) several efficient data structures (most notably Lemma 3.5). Then (Section 4), for each of the considered operations, we make several combinatorial observations, allowing us to restrict the search space of our algorithms, and creating a framework where our data structures can be used efficiently.

Finally (in Section 5), we give algorithms running in $(n \log^{O(1)} \sigma)$-time computing the minimum number of insertions (respectively, substitutions) we need to apply to w in order to obtain a k-universal word, with $k > \iota(w)$. These algorithms rely heavily on the fact that computing the edit distance to k-universality can be reformulated, in this case, as computing the path of length k of minimum weight in a weighted DAG with the Monge property. In particular, these algorithms provide optimal linear-time solutions for our problem in the case of increasing the universality-index of words over constant-size alphabets.

For space reasons, some proofs, examples, and pseudocode for the algorithms are given in the full version of this paper [19]. A discussion on lower bounds is also given in [19].

2 Preliminaries

Let $\mathbb{N}$ be the set of natural numbers and $\mathbb{N}_0 = \mathbb{N} \cup \{0\}$. Define for $i, j \in \mathbb{N}_0$ with $i < j$ the interval $[i : j]$ as $\{i, i+1, \ldots, j-1, j\}$. An alphabet Σ is a nonempty finite set of symbols called *letters*. A *word* is a finite sequence of letters from Σ, thus an element of the free monoid Σ^*. Let $\Sigma^+ = \Sigma^* \backslash \{\varepsilon\}$, where ε is the empty word. The *length* of a word $w \in \Sigma^*$ is denoted by $|w|$. Let Σ^k be the set of all words from Σ^* of length exactly k. A word $u \in \Sigma^*$ is a *factor* of $w \in \Sigma^*$ if $w = xuy$ for some $x, y \in \Sigma^*$. If $x = \varepsilon$ (resp. $y = \varepsilon$), u is called a *prefix* (resp. *suffix*) of w. The i^{th} letter of $w \in \Sigma^*$ is denoted by $w[i]$ for $i \in [1 : |w|]$. Set $w[i : j] = w[i]w[i+1] \cdots w[j]$ for $1 \leq i \leq j \leq |w|$, $|w|_{\mathtt{a}} = |\{i \in [1 : |w|] \,|\, w[i] = \mathtt{a}\}|$, and $\mathrm{alph}(w) = \{\mathtt{a} \in \Sigma \,||w|_{\mathtt{a}} > 0\}$ for $w \in \Sigma^*$. We can now introduce the notion of subsequence.

▶ **Definition 2.1.** *A word $v = v_1 \cdots v_\ell \in \Sigma^*$ is a* subsequence *of $w \in \Sigma^*$ if there exist $x_1, \ldots, x_{\ell+1} \in \Sigma^*$ with $w = x_1 v_1 \cdots x_\ell v_\ell x_{\ell+1}$. Let $\mathrm{Subseq}(w)$ be the set of all subsequences of w and define $\mathrm{Subseq}_k(w) = \mathrm{Subseq}(w) \cap \Sigma^k$, the set of subsequences of w of length $k \in \mathbb{N}$.*

For $k \in \mathbb{N}_0$, $\mathrm{Subseq}_k(w)$ is called the k-spectrum of w. Simon [61] defined the congruence $\sim_k$ in which $u, v \in \Sigma^*$ are congruent if they have the same k-spectrum. As introduced in [6] the notion of k-universality of a word over Σ denotes its property of having Σ^k as k-spectrum.

▶ **Definition 2.2.** *A word $w \in \Sigma^*$ is called* k-subsequence universal *(w.r.t. Σ, for short* k-universal*), for $k \in \mathbb{N}$, if $\mathrm{Subseq}_k(w) = \Sigma^k$. We abbreviate 1-universal by* universal*. The* universality-index *$\iota(w)$ of $w \in \Sigma^*$ is the largest k such that w is k-universal.*

If $\iota(w) = k$ then w is ℓ-universal for all $\ell \leq k$. Notice that k-universality is always w.r.t. a given alphabet Σ: the word $\mathtt{abcba}$ is universal for $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ but it is not universal for $\Sigma \cup \{\mathtt{d}\}$. In each algorithm presented in this paper, whenever we discuss about the universality index of some word (factor of the input word, or obtained from the input word via edit operations), we compute it with respect to the alphabet of the input word w.

The notion of ℓ-universality coincides to that of ℓ-richness introduced in [40, 41]. We use the name *ℓ-universality* rather than *ℓ-richness*, as richness of words is also used with other meanings, see, e.g., [23, 22]. We recall the arch factorisation, introduced by Hebrard [34].

▶ **Definition 2.3** ([34]). *For $w \in \Sigma^*$ the* arch factorisation *of w is $w = \mathrm{ar}_w(1) \cdots \mathrm{ar}_w(k) r(w)$ for some $k \in \mathbb{N}_0$ where $\mathrm{ar}_w(i)$ is universal, the last letter of $\mathrm{ar}_w(i)$, namely $\mathrm{ar}_w(i)[|\mathrm{ar}_w(i)|]$, does not occur in $\mathrm{ar}_w(i)[1 : |\mathrm{ar}_w(i)| - 1]$ for all $i \in [1 : k]$, and alph$(r(w)) \subset \Sigma$. The words $\mathrm{ar}_w(i)$ are called* arches *of w, $r(w)$ is called the* rest.

If the arch factorisation of w contains $k \in \mathbb{N}_0$ arches, then $\iota(w) = k$. The following immediate theorem based on the work of Simon [61] completely characterises the set of k-subsequence universal words, based on Hebrard's arch factorisation.

▶ **Theorem 2.4.** *The word $w \in \Sigma^*$ is k-universal if and only if there exist the words v_i, with $i \in [1 : k]$, such that $v_1 \cdots v_k = w$ and alph$(v_i) = \Sigma$ for all $i \in [1 : k]$.*

General algorithmic framework. The further preliminaries regard algorithms. The computational model we use is the standard unit-cost RAM with logarithmic word size: for an input of size n, each memory-word can hold $\log n$ bits. In all the problems, we assume that we are given a word w, with $|w| = n$, over an alphabet $\Sigma = \{1, 2, \ldots, \sigma\}$, with $|\Sigma| = \sigma \leq n$. This is a common assumption in string algorithms: the input alphabet is said to be *an integer alphabet*. For a more detailed general discussion on this model see, e.g., [17] or the full version of our paper [19]. We also assume that our input words contain at least two distinct letters, otherwise all the problems we consider become trivial.

The following theorem was proven in [6] and shows that the universality index and the arches can be obtained in linear time w.r.t. the word length.

▶ **Theorem 2.5.** *Let w be a word, with $|w| = n$, alph$(w) = \Sigma$, and $\Sigma = \{1, 2, \ldots, \sigma\}$. We can compute in linear time $O(n)$ the arch factorisation of w, and, as such, $\iota(w)$.*

More precisely, one can compute greedily, in linear time, the following decomposition of w into arches $w = u_1 \cdots u_k$ as follows:

- u_1 is the shortest prefix of w with alph$(u_1) = \Sigma$, or $u_1 = w$ if there is no such prefix;
- if $u_1 \cdots u_i = w[1 : t]$, for some $i \in [1 : k]$ and $t \in [1 : n]$, we compute u_{i+1} as the shortest prefix of $w[t + 1 : n]$ with alph$(u_{i+1}) = \Sigma$, or $u_{i+1} = w[t + 1 : n]$ if there is no such prefix.

In our results, we will use two well-known efficient data structures.

First, the *interval union-find* data structure [26, 37].

▶ **Definition 2.6** (Interval union-find). *Let $V = [1 : n]$ and S a set with $S \subseteq V$. The elements of $S = \{s_1, \ldots, s_p\}$ are called borders and are ordered $0 = s_0 < s_1 < \ldots < s_p < s_{p+1} = n + 1$ where s_0 and s_{p+1} are generic borders. For each border s_i, we define $V(s_i) = [s_{i-1} + 1 : s_i]$ as an induced interval. Now, $P(S) := \{V(s_i) \mid s_i \in S\}$ gives an ordered partition of the set V. The* interval union-find structure *maintains the partition $P(S)$ under the operations:*

- *For $u \in V$, `find`(u) returns $s_i \in S \cup \{n + 1\}$ such that $u \in V(s_i)$.*
- *For $u \in S$, `union`(u) updates the partition $P(S)$ to $P(S \setminus \{u\})$. That is, if $u = s_i$, then we replace the intervals $V(s_i)$ and $V(s_{i+1})$ by the single interval $[s_{i-1} + 1 : s_{i+1}]$ and update the partition so that further `find` and `union` operations can be performed.*

When using the data structure from Definition 2.6, we employ a less technical language: we describe the intervals stored initially in the structure, and then the unions are made between adjacent intervals. We can enhance the data structures so that the `find` operation returns both borders of the interval containing the searched value, as well as some other satellite data we decide to associate to that interval. The following lemma was shown in [26, 37].

▶ **Lemma 2.7.** *One can implement the interval union-find data structure, such that, the initialisation of the structures followed by a sequence of $m \in O(n)$ union and find operations can be executed in $\mathcal{O}(n)$ time and space.*

Finally, we recall the *Range Minimum Query* problem, and the main result on it [8].

▶ **Definition 2.8** (RMQ)**.** *Let A be an array with n elements from a well-ordered set. We define* range minimum queries RMQ_A *for the array of A:* $\mathrm{RMQ}_A(i,j) = \arg\min\{A[t] \mid t \in [i:j]\}$, *for $i,j \in [1:n]$. That is, $\mathrm{RMQ}_A(i,j)$ is the position of the smallest element in the subarray $A[i:j]$; if there are multiple positions containing this smallest element, $\mathrm{RMQ}_A(i,j)$ is the leftmost of them. (When it is clear from the context, we drop the subscript A).*

▶ **Lemma 2.9.** *Let A be an array with n integer elements. One can preprocess A in $O(n)$ time and produce data structures allowing to answer in constant time* range minimum queries $\mathrm{RMQ}_A(i,j)$, *for any $i,j \in [1:n]$.*

3 Toolbox

In this section we present data structures which will be decisive in obtaining efficient solutions for the approached problems. Our running example will be the word $w = \texttt{bananaban}$, on which we illustrate some of the notions we define here. Full details are given in [19].

For a word w over an alphabet Σ, a position j of w, and a letter $a \in \Sigma$ which occurs in $w[1:j]$, let $\mathrm{last}_j[a] = \max\{i \leq j \mid w[i] = a\}$, the last position where a occurs before j; if a does not occur in $w[1:j]$ or for $j = 0$, then, by convention, $\mathrm{last}_j[a] = |w| + 1$. Let $S_j = \{\mathrm{last}_j[a] \mid a \in \mathrm{alph}(w[1:j])\}$. If i,j are two positions of w, let $\Delta(i,j)$ be the number of distinct letters occurring in $w[i:j]$, i.e., $\Delta(i,j) = |\mathrm{alph}(w[i:j])|$; if $i > j$, then $\Delta(i,j) = 0$. For a position i of w, and a letter $a \in \Sigma$, let $d_i[a] = \Delta(\mathrm{last}_i[a], i)$.

▶ **Lemma 3.1.** *Let w be a word, with $|w| = n$, $alph(w) = \Sigma$, and $\Sigma = \{1, 2, \ldots, \sigma\}$. We can compute in $O(n)$ the values $\Delta(1,\ell)$, for all $\ell \in [1:n]$.*

▶ **Lemma 3.2.** *Let w be a word, with $|w| = n$, $alph(w) = \Sigma$, and $\Sigma = \{1, 2, \ldots, \sigma\}$. We can compute in $O(n)$ the values $\Delta(i - \sigma + 1, i)$, for all $i \in [\sigma : n]$.*

▶ **Lemma 3.3.** *Let w be a word, with $|w| = n$, $alph(w) = \Sigma$, and $\Sigma = \{1, 2, \ldots, \sigma\}$. We can compute in $O(n)$ time $\mathrm{last}_{j\sigma+1}[a]$ and $d_{j\sigma+1}[a]$, for all $a \in \Sigma$ and all integers $1 \leq j \leq (n-1)/\sigma$.*

For $w = \texttt{bananaban}$, we have $|w| = 9$ and $\sigma = 3$. In Lemma 3.1 we compute $\Delta(1,1) = 1$, $\Delta(1,2) = 2$, and $\Delta(1,\ell) = 3$ for $\ell \in [3:9]$. In Lemma 3.2 we compute $\Delta(1,3) = 3$, $\Delta(2,4) = \Delta(3,5) = \Delta(4,6) = 2$, $\Delta(5,7) = 3$, $\Delta(6,8) = 2$, and $\Delta(7,9) = 3$. In Lemma 3.3 we compute the arrays $\mathrm{last}_4[\cdot]$ and $\mathrm{last}_7[\cdot]$. We get: $\mathrm{last}_4[\texttt{a}] = 4$, $\mathrm{last}_4[\texttt{b}] = 1$, $\mathrm{last}_4[\texttt{n}] = 3$, and $\mathrm{last}_7[\texttt{a}] = 6$, $\mathrm{last}_7[\texttt{b}] = 7$, $\mathrm{last}_7[\texttt{n}] = 5$. Therefore, $S_4 = \{1,3,4\}$, $S_7 = \{5,6,7\}$, and $d_4[\texttt{a}] = 1$, $d_4[\texttt{b}] = 3$, $d_4[\texttt{n}] = 2$, $d_7[\texttt{a}] = 2$, $d_7[\texttt{b}] = 1$, $d_7[\texttt{n}] = 3$.

For a word w and a position i of w, let $\mathrm{univ}[i] = \max\{j \mid w[j:i] \text{ is universal}\}$. That is, for the position i we compute the shortest universal word ending on that position. If there is no universal word ending on position i we set $\mathrm{univ}[i] = 0$.

Further, if $n = |w|$, let $V_w = \{\mathrm{univ}[i] \mid 1 \leq i \leq n\}$. In V_w we collect the starting positions of the shortest universal words ending at each position of the word w. Now, for $j \in V_w$, let $L_j = \{i \mid \mathrm{univ}[i] = j\}$; in other words, we group together the positions i of w for which the shortest universal word ending on i starts on some position j. Note that $L_0 = \{i \mid w[1:i] \text{ is not universal}\}$, i.e., the positions of w where no universal word ends.

Several observations are immediate: for $i \in L_j$, $i' \in L_{j'}$, we have $i \leq i'$ if and only if $j \leq j'$. As each position i of w belongs to a set L_j, for some $j \in V_w$, we get that $\{L_j \mid j \in V_w\}$ is a partition of $[1 : n]$ into intervals. Furthermore, $w[i] \neq w[j]$ for all $i \in L_j$ and $j \neq 0$: if $w[i]$ would be the same as $w[j]$ then $w[j+1 : i]$ would also be a universal word, so i would not be in L_j. Also, if $i = \max(L_j)$ for some $j > 0$ then $w[i+1] = w[j]$. Indeed, there exists $j' \in [j+1 : i]$ such that $w[j' : i+1]$ is universal. But $w[j]$ does not occur in $w[j' : i]$, so $w[j] = w[i+1]$ must hold.

Further, we define for all positions i of w the value $\text{freq}[i] = |w[1 : i]|_{w[i]}$, the number of occurrences of $w[i]$ in $w[1 : i]$. Also, let $T[i] = \min\{|w[i+1 : n]|_a \mid a \in \Sigma\}$, for $i \in [0, n-1]$, be the least number of occurrences of a letter in $w[i+1 : n]$; set $T[n] = 0$.

▶ **Lemma 3.4.** *Let w be a word, with $|w| = n$, $alph(w) = \Sigma$, and $\Sigma = \{1, 2, \ldots, \sigma\}$. We can compute in $O(n)$ time the following data structures:* 1. *the array* $\text{univ}[\cdot]$*;* 2. *the set V_w and the lists L_j, for all $j \in V_w \setminus \{0\}$;* 3. *the array* $\text{freq}[\cdot]$*;* 4. *the array* $T[\cdot]$*;* 5. *the values* $\text{last}_{j-1}[w[i]]$*, for all $j \in V_w$ and all $i \in L_j$;* 6. *the values* $\text{last}_{i-1}[w[i]]$*, for all $i \in [2 : n]$.*

Consider again $w = \texttt{bananaban}$. In Lemma 3.4 we compute the following values. Firstly, $\text{univ}[1] = \text{univ}[2] = 0$, $\text{univ}[\ell] = 1$ for $\ell \in [3 : 6]$, $\text{univ}[7] = \text{univ}[8] = 5$, $\text{univ}[9] = 7$. Thus, $V_w = \{0, 1, 5, 7\}$ and $L_0 = [1 : 2]$, $L_1 = [3 : 6]$, $L_5 = [7 : 8]$, $L_7 = [9 : 9]$. Secondly, $\text{freq}[1] = 1$, $\text{freq}[2] = \text{freq}[3] = 1$, $\text{freq}[4] = \text{freq}[5] = 2$, $\text{freq}[6] = 3$, $\text{freq}[7] = 2$, $\text{freq}[8] = 4$, $\text{freq}[9] = 3$. Moreover, $T[0] = 2$, $T[\ell] = 1$ for $\ell \in [1 : 6]$, and $T[\ell] = 0$ for $\ell \in [7 : 9]$. Then, for $j = 1$, we have $\text{last}_0[a] = 0$, for $a \in \{\texttt{a}, \texttt{b}, \texttt{n}\}$; for $j = 5$, we have $\text{last}_4[\texttt{b}] = 1$ and $\text{last}_4[\texttt{a}] = 4$; for $j = 7$, we have $\text{last}_6[\texttt{n}] = 5$. Finally, $\text{last}_0[\texttt{b}] = 10$, $\text{last}_1[\texttt{a}] = 10$, $\text{last}_2[\texttt{n}] = 10$, $\text{last}_3[\texttt{a}] = 2$, $\text{last}_4[\texttt{n}] = 3$, $\text{last}_5[\texttt{a}] = 4$, $\text{last}_6[\texttt{b}] = 1$, $\text{last}_7[\texttt{a}] = 6$, $\text{last}_8[\texttt{n}] = 5$.

The main idea behind proving Lemmas 3.1, 3.3, and 3.4 is to traverse the word w left to right (or, respectively, right to left) and maintain the number of occurrences, as well as the last occurrence, of each letter in the prefix (respectively, suffix) of w that we have visited so far. For Lemma 3.2, we only consider a sliding window of size σ which traverses the word left-to-right, while maintaining similar data as before, but only for the content of the window. In all cases, this requires linear time and enables us to construct the desired data structures.

Together with the string-processing data structures we defined above, we need the following general technical data structures lemma. This lemma (combined with some combinatorial observations) will be used to speed up some of our dynamic programming algorithms.

In this lemma we process a list A which initially has σ elements, and in which we insert, in successive steps, σ new elements, by appending them always at the same end. For simplicity, we can assume that the list A is a sequence with 2σ elements (denoted $A[i]$, with $i \in [1 : 2\sigma]$), out of which the last σ are initially undefined. The i^{th} insertion would, consequently, mean setting $A[\sigma + i]$ to the actual value that we want to insert in the list A. In our lemma we will also repeatedly perform an operation which decrements the values of some elements of the list A. However, we will not require to be able to explicitly access, after every operation, all the elements of the list (so we will not need to retrieve the values $A[i]$). Consequently, we will not maintain explicitly the value of all the elements of A (that is, we will not update the elements affected by decrements). We are only interested in being able to retrieve (by value and position), at each moment, the smallest element and the last element of A. Thus, throughout the computation, we only maintain a subset of important elements of A, including the aforementioned two. We can now state our result, whose proof is based on Lemma 2.7.

▶ **Lemma 3.5.** *Let A be a list with σ elements (natural numbers) and let $m = \sigma$. We can execute (in order) the sequence of σ operations $o_1, \ldots, o_\sigma$ on A in overall $O(\sigma)$ time, where o_i consists of the following three steps, for $i \in [1 : m]$:*

1. *Return* $e = \arg\min\{A[i] \mid i \in [1:m]\}$ *and* $A[e]$.
2. *For some* $j_i \in [1:m]$, *decrement all elements* $A[j_i], A[j_i+1], \ldots, A[m]$ *by* 1.
3. *For some natural number* x_i, *append the element* x_i *to* A *(i.e., set* $A[m+1]$ *to* x_i*), and increment* m *by* 1 *(i.e., set* m *to* $m+1$*).*

Proof. Firstly, we will *run a preprocessing* of A.

We begin by defining recursively a finite sequence of positions as follows:

- a_1 is the rightmost position of A on which $\min\{A[i] \mid i \in [1:\sigma]\}$ occurs;
- for $i \geq 2$, if $a_{i-1} < \sigma$, then a_i is the rightmost position on which $\min\{A[i] \mid i \in [a_{i-1}+1 : \sigma]\}$ occurs;
- for $i \geq 2$, if $a_{i-1} = \sigma$, then we can stop, our sequence will have $i-1$ elements.

Let p be the number of elements in the sequence defined above, i.e., our sequence is $a_1, \ldots, a_p$. For convenience, let $a_0 = 0$. Then the sequence $a_1, \ldots, a_p$ fulfils the following properties:

- $a_p = \sigma$ and $a_i > a_{i-1}$, for all $i \in [1:p]$;
- $A[a_i] > A[a_{i-1}]$ for all $i \in [2:p]$;
- for all $i \in [1:p]$, we have $A[a_i] < A[t]$, for all $t \in [a_i+1:\sigma]$;
- for all $i \in [1:p]$, we have $A[a_i] \leq A[t]$, for all $t \in [a_{i-1}+1:a_i]$.

By definition, for $i \in [1:p]$ we have $A[a_i] = \min\{A[t] \mid t \in [a_{i-1}+1:\sigma]\}$, $A[a_i] < \min\{A[t] \mid t \in [a_i+1:\sigma]\}$, and $a_1 = \min\{A[i] \mid i \in [1:\sigma]\}$. Clearly, we have $a_p = \sigma$.

The positions $a_1, \ldots, a_p$ can be computed in linear time $O(\sigma)$, in reversed order. As we do not know from the beginning the value of p, we will compute a sequence $b_1, b_2, \ldots$ of positions as follows. We start with $b_1 = \sigma$, $t = \sigma - 1$, and $i = 2$. Then, while $t \geq 1$ we do the following case analysis. If $A[t] < b_{i-1}$, then set $b_i = t$, increment i by 1, and decrement t by 1. Otherwise, if $A[t] \geq b_{i-1}$, just decrement t by 1. It is straightforward that this process takes $O(\sigma)$ time, and, when we have finished it, the number i is exactly the number p, and $a_i = b_{p-i+1}$.

Another observation is that, for $a_0 = 0$, the intervals $[a_{i-1}+1, a_i]$, for $i \in [1,p]$, define a partition of the interval $[1:\sigma]$ into p intervals. Therefore, we can define a partition of the interval $[1:2\sigma]$ into the intervals $[a_{i-1}+1:a_i]$, for $i \in [1,p]$, and $[t:t]$, for $t \in [\sigma+1:2\sigma]$. Thus, we construct in linear time, according to Lemma 2.6, an interval union-find data structure for the interval $[1:2\sigma]$, as induced by the intervals $[1:a_1]$, $[a_1+1:a_2], \ldots$ $[a_{p-1}, a_p]$, $[\sigma+1:\sigma+1]$, $[\sigma+2:\sigma+2], \ldots$ $[2\sigma:2\sigma]$.

Let us now take $m = \sigma$ (and assume the convention $A[0] = 0$). We associate as satellite data to each interval $[x:y]$ with $y \leq m$ from our interval union-find data structure the value $A[y] - A[x-1]$.

This entire preprocessing takes clearly $O(\sigma)$ time.

In order to explain how the operations are implemented, we assume as invariant that the following properties are fulfilled before o_i is executed, for $i \in [1:\sigma]$:

- A contains m elements;
- all intervals $[x:y]$ with $y > m$ from our interval union-find data structure are singletons (i.e., $x = y$);
- for each interval $[x:y]$ with $y \leq m$, we have the associated satellite data $A[y] - A[x-1]$;
- for each interval $[x:y]$ with $y \leq m$, we have that $A[y] \leq A[t]$ for $t \in [x:m]$ and $A[y] < A[t]$ for $t \in [y+1:m]$;
- we have stored in a variable ℓ the value $A[m]$.

This clearly holds after the preprocessing step, so before executing o_1.

Let us now explain *how the operation o_i is executed.*

The first step of o_i is to return $e = \min\{A[i] \mid i \in [1 : m]\}$ and i_e the rightmost position of the list A such that $A[i_e] = e$. We execute `find`(1) to return the first interval $[1 : i_e]$ stored in our interval union-find data structure; $A[i_e]$ is the satellite data associated to this interval (by convention, $A[i_e] - A[1 - 1] = A[i_e] - A[0] = A[i_e]$). The fact that the invariant property holds shows that i_e is correctly computed.

The second step of o_i is to decrement all elements $A[j_i], A[j_i + 1], \ldots, A[m]$ by 1, for some $j_i \in [1 : m]$. We will make no actual change to the elements of the list A, as this would be too inefficient, but we might have to change the state of the union-find data structure, as well as the satellite data associated to some intervals of this structure.

So, let $[x : y]$ be the interval containing j_i, returned by `find`(j_i), and also assume first that $x \neq 1$.

According to the invariant, $A[j_i] \geq A[y]$ and $A[y] > A[x - 1]$. After decrementing the elements $A[j_i], A[j_i + 1], \ldots, A[m]$ by 1, the difference $A[t] - A[t']$ is exactly the same as before, for all $t, t' \in [j_i : m]$. In consequence, the relative order between the elements of the suffix $A[j_i : m]$ of the list A is preserved. Also, for all $t \in [x : j_i - 1]$, we have now $A[t] > A[y]$ (before decrementing $A[y]$ we had only $A[t] \geq A[y]$). However, the difference $A[y] - A[x - 1]$ is now decreased by 1. If it stays strictly positive, we just update the satellite data of the respective interval (by decrementing it accordingly by 1). If $A[y] - A[x - 1] = 0$, then we make the `union` of the interval $[z : x - 1]$ (returned by `find`$(x - 1)$) and $[x : y]$ to obtain the new interval $[z : y]$. Its satellite data is $A[y] - A[z - 1] = A[x - 1] - A[z - 1]$, so the same as the satellite data that was before associated to $[z : x - 1]$. The invariant is clearly preserved, as, even after decrementing it, $A[y]$ (which is now equal to $A[x - 1]$) is strictly greater than $A[z - 1]$, strictly smaller than $A[t]$, for $t \in [y + 1 : m]$, and smaller than or equal to $A[t]$, for $t \in [z : y]$.

If the interval containing j_i is $[1 : y]$, then we just update the satellite data of the respective interval by decrementing it by 1.

The third step of o_i is to append the element x_i to A (i.e., set $A[m + 1] = x_i$), for some natural number x_i, and increment m by 1.

We implement this as follows. Let $t = m$ and $q = A[m]$ (this value is stored and maintained using the variable ℓ). While $t \geq 1$ do the following. Let $[z, t]$ be the interval returned by `find`(t); we have $q = A[t]$. If $q \geq x_i$, make the union of $[z : t]$ and $[t + 1 : m + 1]$; update $q = q - (A[t] - A[z - 1]) = A[z - 1]$ (using the satellite data $A[t] - A[z - 1]$ associated to $[z, t]$), update $t = z - 1$, and reiterate the loop. If $q < x_i$, exit the loop. After this, we set m to $m + 1$ and $\ell = x_i$.

It is not hard to see that after running this third step, so before executing operation o_{i+1}, the invariant is preserved.

Performing operation o_i takes an amount of time proportional to the sum of the number of `union` and the number of `find` operations executed during its three steps. By Lemma 2.7, this means that executing all operations $o_1, \ldots, o_\sigma$ takes in total at most $O(\sigma)$ time. ◀

4 Edit Distance

We are interested in computing the minimal number of edits we need to apply to a word w, with $|w| = n$, $\mathrm{alph}(w) = \Sigma$, with universality index $\iota(w)$, so that it is transformed into a word with universality index k, w.r.t. the same alphabet Σ. The edits considered are insertion, deletion, substitution, and the number we want to compute can be seen as the *edit distance* between w and the set of k-universal words over Σ.

However, if we want to obtain a k-universal word with $k > \iota(w)$, then it is enough to consider only insertions. Indeed, deleting a letter of a word can only restrict the set of subsequences of the respective word, while in this case we are interested in enriching it. Substituting a letter might make sense, but it can be simulated by an insertion: assume one wants to substitute the letter $\mathtt{a}$ on position i of a word w by a $\mathtt{b}$. It is enough to insert a $\mathtt{b}$ next to position i, and the set of subsequences of w is enriched with all the words that could have appeared as subsequences of the word where $\mathtt{a}$ was actually replaced by $\mathtt{b}$. We might have some extra words in the set of subsequences, which would have been eliminated through the substitution, but it does not affect our goal of reaching k-universality.

If we want to obtain a word with universality index k, for $k < \iota(w)$, then it is enough to consider only deletions. Assume that we have a sequence of edits that transforms the word w into a word w' with universality index k. Now, remove all the insertions of letters from that sequence. The word w'' we obtain by executing this new sequence of operations clearly fulfils $\iota(w'') \leq \iota(w')$. Further, in the new sequence, replace all substitutions with deletions. We obtain a word w''' with a set of subsequences strictly included in the one of w'', so with $\iota(w''') \leq \iota(w'')$. As each deletion changes the universality index by at most 1, it is clear that (a prefix of) this new sequence of deletions witnesses a shorter sequence of edits which transforms w into a word of universality index k.

So, to increase the universality index of a word it is enough to use insertions and to decrease the universality index of a word it is enough to use deletions. Nevertheless, one might be interested in what happens if we only use substitutions. In this way, we can both decrease and increase the universality index of a word. Moreover, one can see the minimal number of substitutions needed to transform w into a k-universal word as the Hamming distance between w and the set of k-universal words. We will discuss each of these cases separately.

4.1 Insertions

▶ **Theorem 4.1.** *Let w be a word, with $|w| = n$, $alph(w) = \Sigma$, and $\Sigma = \{1, 2, \ldots, \sigma\}$. Let $k \geq \iota(w)$ be an integer. We can compute the minimal number of insertions needed to apply to w in order to obtain a k-universal word (w.r.t. Σ) in $O(nk)$ time if $k \leq n$ and $O(T(n, \sigma, k))$ time otherwise, where $T(n, \sigma, k)$ is the time needed to compute $k\sigma - n$.*

Proof.

Case 1. Let us assume first that $k \leq n$. We structured our proof in such a way that the idea of the solution, as well as the actual computation steps, and the arguments supporting their correctness are clearly marked. Pseudocode for this algorithm is given in the full version of the paper [19].

§ General approach. We want to transform the word w into a k-universal word with a minimal number of insertions. Assume that the word we obtain this way is w', and $|w'| = m$. Thus, w' has a prefix $w'[1 : m']$ which is k-universal, but $w'[1 : m' - 1]$ is not k-universal. Moreover, $w'[1 : m']$ is obtained from a prefix $w[1 : \ell]$ of w, and $w'[m'+1 : m] = w[\ell+1 : n]$. Indeed, any insertion done to obtain $w'[m' + 1 : m]$ can be simply omitted and still obtain a k-universal word from w, with a lower number of insertions.

Consequently, it is natural to compute the minimal number of insertions needed to transform $w[1 : \ell]$ into a t-universal word, for all $\ell \leq n$ and $t \leq k$. Let $M[\ell][t]$ denote this number. By the same reasoning as above, transforming (with insertions) $w[1 : \ell]$ into a t-universal word means that there exists a prefix $w[1 : \ell']$ of $w[1 : \ell]$ which is transformed into a $(t - 1)$-universal word and $w[\ell' + 1 : \ell]$ is transformed into a 1-universal word. Clearly, the number of insertions needed to transform $w[\ell'+1 : \ell]$ into a 1-universal word is $\sigma - \Delta(\ell'+1, \ell)$, i.e., the number of distinct letters not occurring in $w[\ell'+1:\ell]$. As we are interested in the minimal number of insertions needed to transform $w[1:\ell]$ into a t-universal word, we need to find a position ℓ' such that the total number of insertions needed to transform $w[1:\ell']$ into a $(t-1)$-universal word and $w[\ell'+1:\ell]$ into a 1-universal word is minimal.

§ Algorithm – initial idea. So, for $\ell \in [1:n]$ and $t \in [1:k]$, $M[\ell][t]$ is the minimal number of insertions needed to make $w[1:\ell]$ t-universal. By the explanations above, we get the following recurrence $M[\ell][t] = \min\{M[\ell'][t-1] + (\sigma - \Delta(\ell'+1, \ell)) \mid \ell' \leq \ell\}$. Clearly, $M[\ell][1] = \sigma - \Delta(1, \ell)$. Also, it is immediate to note that $M[\ell][t] \geq M[\ell''][t]$ for all $\ell \leq \ell''$. Indeed, transforming a word into a t-universal word can always be done with at most as many insertions as those used in transforming any of its prefixes into a t-universal word.

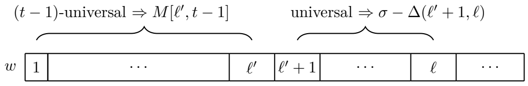
$(t-1)$-universal ⇒ $M[\ell', t-1]$
universal ⇒ $\sigma - \Delta(\ell'+1, \ell)$
w
1
$\cdots$
ℓ'
$\ell'+1$
$\cdots$
ℓ
$\cdots$

Figure 1 Illustration of the formula developed for the computation of $M[\ell][t]$.

We now want to compute the elements of matrix M. Before this, we produce the data structures of Lemma 3.3 (and we use the notations from its framework). That is, we compute in $O(n)$ time $\mathrm{last}_{j\sigma+1}[a]$ and $d_{j\sigma+1}[a] = \Delta(\mathrm{last}_{j\sigma+1}[a], j\sigma+1)$, for all $a \in \Sigma$ and all $j \leq \frac{(n-1)}{\sigma}$.

By Lemma 3.1, we can compute the values $M[\ell][1]$, for all $\ell \in [1:n]$ in $O(n)$ time. However, a direct computation of the values $M[\ell][t]$, for $t > 1$, according to the recurrence above is not efficient. Implemented directly, it requires $O(n^2k)$ time; using an efficient structure (e.g., interval trees) for computing the various minima leads to an $O(nk \log n)$-time solution; exploiting the algebraic properties of M (related to the Monge property [1]) leads to an $O(nk \log \sigma / \log \log \sigma)$-time solution. We will describe a more efficient solution.

§ A useful observation. Assume that to transform $w[1:\ell]$ into a t-universal word we transform $w[1:\ell']$ into a $(t-1)$-universal word and $w[\ell'+1:\ell]$ into a 1-universal word. The number of insertions needed to do this is $M[\ell'][t-1] + (\sigma - \Delta(\ell'+1, \ell))$. If $w[\ell'+1]$ occurs twice in $w[\ell'+1:\ell]$, then $M[\ell'][t-1] + (\sigma - \Delta(\ell'+1, \ell)) \geq M[\ell'+1][t-1] + (\sigma - \Delta(\ell'+2, \ell))$. Thus, we can rewrite our recurrence in the following way, using the framework of Lemma 3.3: $M[\ell][t] = \min\{M[\ell'][t-1] + (\sigma - \Delta(\ell'+1, \ell)) \mid \ell'+1 \in S_\ell \cup \{\ell+1\}\}$ (recall the definition of $S_\ell = \{\mathrm{last}_\ell[a] \mid a \in \mathrm{alph}(w[1:\ell])\}$ from Section 3).

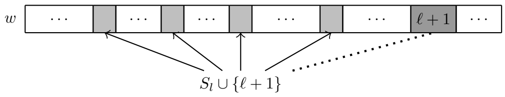

Figure 2 Only the positions $\ell'+1 \in \{\mathrm{last}_\ell[a] \mid a \in \mathrm{alph}(w[1:\ell])\} \cup \{\ell+1\} = S_\ell \cup \{\ell+1\}$ are needed to compute $M[\ell][t]$ by dynamic programming. These positions are depicted here in grey.

Once more, a brief analysis can be done. Using directly this observation leads to an $O(nk\sigma)$-time algorithm for our problem; an implementation based on, e.g., interval trees runs in $O(nk \log \sigma)$-time. In the following we see that a faster solution exists.

In fact, in the efficient version of our algorithm we will use a slightly weaker formula, where the minimum is computed for all elements $\ell'+1$ from a set $S'_\ell \cup \{\ell+1\}$, instead of the set $S_\ell \cup \{\ell+1\}$, where S'_ℓ is a superset of size at most 2σ of S_ℓ defined as follows. If $\ell = j\sigma + i$, for some $j \leq (n-1)/\sigma$ and $i \in [1:\sigma]$, then $S'_\ell = \begin{cases} S_\ell & \text{if } i = 1, \\ S_{j\sigma+1} \cup \{j\sigma+2, \ldots, j\sigma+i\} & \text{if } i \in [2:\sigma]. \end{cases}$

§ Algorithm – the efficient variant. Using the observation above, together with Lemma 3.5, we can compute the elements of the matrix M efficiently using dynamic programming.

So, let us consider a value $t \geq 2$. Assume that we have computed the values $M[\ell][t-1]$, for all $\ell \in [1:n]$. We now want to compute the values $M[\ell][t]$, for all $\ell \in [1:n]$. The main idea in doing this efficiently is to split the computation of the elements on column $M[\cdot][t]$ of the matrix M in phases. In phase j we compute the values $M[j\sigma+1][t]$, $M[j\sigma+2][t], \ldots,$ $M[(j+1)\sigma][t]$, for $j \leq (n-1)/\sigma$.

We now consider some j, with $0 \leq j \leq (n-1)/\sigma$. We want to apply Lemma 3.5, so we need to define the list A of size σ. This is done as follows.

We will maintain an auxiliary array $\mathrm{pos}[\cdot]$ with σ elements. Moreover, the element $A[i]$, for each i, is accompanied by two satellite information: a position of w and the letter found on that position. For a from 1 to σ, if $d_{j\sigma+1}[a] = \sigma - i$ for some $i < \sigma$ then we set $A[i+1] = M[\mathrm{last}_{j\sigma+1}[a]-1][t-1]+i$ and $\mathrm{pos}[a] = i+1$; the satellite data of $A[i+1]$ is the pair $(\mathrm{last}_{j\sigma+1}[a], a)$. If, for some letter a, $\mathrm{last}_{j\sigma+1}[a] = n+1$ and $d_{j\sigma+1}[a] = 0$ (i.e., a does not occur in $w[1:j\sigma+1]$) we set $\mathrm{pos}[a] = 0$.

Intuitively, one can see the elements of A as triples: $(A[e], \mathrm{last}_{j\sigma+1}[a], a)$ where $A[e] = M[\mathrm{last}_{j\sigma+1}[a]-1][t-1]+e-1$, with $e \in [1:\sigma]$ and $a \in \Sigma$. More precisely, let $a_d, a_{d-1}, \ldots, a_1$ be the letters of Σ that occur in $w[1:j\sigma+1]$, ordered such that $\mathrm{last}_{j\sigma+1}[a_e] < \mathrm{last}_{j\sigma+1}[a_f]$ if and only if $e > f$. At this point, we have defined only the last d elements of A and, for $i \in [1:d]$, the element on position $\sigma-i+1$ is $A[\sigma-i+1] = M[\mathrm{last}_{j\sigma+1}[a_i]-1][t-1]+(\sigma-i)$ and has the satellite data $(\mathrm{last}_{j\sigma+1}[a_i], a_i)$. Also, $\mathrm{pos}[a_i] = \sigma-i+1$. The first $\sigma-d$ elements of A are set to ∞; as convention, applying arithmetic operations to ∞ leaves it unchanged.

We set m to σ and define (and apply) a sequence of operations $o_1, \ldots, o_\sigma$ as in Lemma 3.5.

An invariant: We want to ensure that the list A fulfils the following invariant properties before the execution of each operation o_i.

- For $e \in [1:d]$, the triple on position $\sigma-e+1$ of A is: $(M[\mathrm{last}_{j\sigma+1}[a_e]-1][t-1]+(\sigma-\Delta(\mathrm{last}_{j\sigma+1}[a_e], j\sigma+i)), \mathrm{last}_{j\sigma+1}[a_e], a_e)$. That is, $A[\sigma-e+1] = M[\mathrm{last}_{j\sigma+1}[a_e]-1][t-1]+(\sigma-\Delta(\mathrm{last}_{j\sigma+1}[a_e], j\sigma+i))$.
- For $g \in [1:i-1]$, the triple on position $\sigma+g$ of A is: $(M[j\sigma+g][t-1]+(\sigma-\Delta(j\sigma+g+1, j\sigma+i)), j\sigma+g+1, w[j\sigma+g+1])$. That is $A[\sigma+g] = M[j\sigma+g][t-1]+(\sigma-\Delta(j\sigma+g+1, j\sigma+i))$.
- $\mathrm{pos}[a]$ is the position of the rightmost position i storing a triple $(A[i], \ell, a)$.

That is, the list A contains all the values $M[\ell][t-1]+(\sigma-\Delta(\ell+1, j\sigma+i))$, for $\ell+1 \in S_{j\sigma+1} \cup \{j\sigma+2, \ldots, j\sigma+i\}$, and $\mathrm{pos}[a]$ indicates the rightmost position of the list A where we store a value $M[\ell][t-1]+(\sigma-\Delta(\ell+1, j\sigma+i))$ with $w[\ell+1] = a$. A consequence of this is that $A[\mathrm{pos}[a]] = M[\mathrm{last}_{j\sigma+i}[a]-1][t-1]+(\sigma-\Delta(\mathrm{last}_{j\sigma+i}[a], j\sigma+i))$.

The invariant clearly holds for $i = 1$.

§ Algorithm – application of Lemma 3.5. In o_i, we extract the minimum q of A. Then set $M[j\sigma+i][t] = \min\{q, M[j\sigma+i][t-1]+\sigma\}$. We decrement by 1 all elements of A on the positions $\mathrm{pos}[a]+1, \mathrm{pos}[a]+2, \ldots, m$, where $a = w[j\sigma+i+1]$. Then, we append to A the element $M[j\sigma+i][t-1]+(\sigma-1)$, with the satellite data $(j\sigma+i+1, a)$, which implicitly increments m by 1, and set $\mathrm{pos}[a] = m$.

▷ Claim 4.2. The invariant holds after operation o_i.

Proof of Claim 4.2. We now need to show that the invariant is preserved after this step. If $a = w[j\sigma+i+1]$ then the number of distinct letters occurring after each position $g > \mathrm{last}_{j\sigma+i}[a]$ in $w[1:j\sigma+i]$ is exactly one smaller than the number of distinct letters occurring after g in $w[1:j\sigma+i+1]$. This means that $M[g-1][t-1]+(\sigma-\Delta(g, j\sigma+i+1))$ is one smaller than

$M[g-1][t-1]+(\sigma-\Delta(g,j\sigma+i))$. Consequently, all values occurring on positions greater than pos$[a]$ in the list A, which stored some values $M[g-1][t-1]+(\sigma-\Delta(g,j\sigma+i+1))$ with $g>\text{last}_{j\sigma+i}[a]$, should be decremented by 1. Also, the number of distinct letters occurring after each position $g\leq\text{last}_{j\sigma+i}[a]$ in $w[1:j\sigma+i]$ is exactly the same as number of distinct letters occurring after g in $w[1:j\sigma+i+1]$. Thus, all values occurring on positions smaller or equal to pos$[a]$ in the list A, which stored some values $M[g-1][t-1]+(\sigma-\Delta(g,j\sigma+i+1))$ with $g\leq\text{last}_{j\sigma+i}[a]$, should stay the same. So, the invariant holds for the first $\sigma+i-1$ positions of A. After appending $M[j\sigma+i][t-1]+(\sigma-1)$ to A and incrementing m, then the invariant holds for the position $\sigma+i$ (which is also the last position) of A too, so the invariant still holds for all positions of A.

Furthermore, the only position of the pos array that needs to be updated after operation o_i is pos$[a]$, and it needs to be set to the new value of m. This is exactly what we do. ◁

▷ Claim 4.3. $M[j\sigma+i][t]$ is correctly computed, for all $i\in[1:\sigma]$.

Proof of Claim 4.3. According to the invariant, before executing operation o_i, A contains the values $M[\ell][t-1]+(\sigma-\Delta(\ell+1,j\sigma+i))$, for $\ell+1\in S_{j\sigma+1}$, and $M[j\sigma+g][t-1]+(\sigma-\Delta(j\sigma+g+1,j\sigma+i))$, for $g\in[1:i-1]$. As $S'_{j\sigma+i}=S_{j\sigma+1}\cup\{j\sigma+g+1\mid g\in[1:i-1]\}$ is a superset of size at most 2σ of $S_{j\sigma+i}$, we obtain that $M[j\sigma+i][t]$ is correctly computed as the minimum between the smallest value in A and $M[j\sigma+i][t-1]+\sigma$. ◁

§ Algorithm – the result of applying Lemma 3.5. After executing the σ operations $o_1,\ldots,o_\sigma$, we have computed the values $M[j\sigma+1][t]$, $M[j\sigma+2][t],\ldots$, $M[(j+1)\sigma][t]$ correctly. We can move on to phase $j+1$ and repeat this process.

§ The result and complexity. The minimal number of insertions needed to make w k-universal is, according to the observations we made, correctly computed as $M[n][k]$.

By Lemma 3.5, computing $M[j\sigma+1][t]$, $M[j\sigma+2][t],\ldots$, $M[(j+1)\sigma][t]$ takes $O(\sigma)$ for each j. Overall, computing the entire column $M[\cdot][t]$ takes $O(n)$ time. We do this for all $t\leq k$, so we use $O(nk)$ time in total to compute all elements of M. This concludes Case 1.

Case 2. If $k>n$, we return $k\sigma-n$. We need, in all cases, $k\sigma-n$ insertions to obtain a word of length $k\sigma$ from w. This is also sufficient: we first use $n(\sigma-1)$ insertions to transform w into $(1\cdots\sigma)^n$; then, by $(k-n)\sigma$ insertions, we further transform it into $(1\cdots\sigma)^k$. So the time needed to solve our problem, in this case, is the time needed to compute $k\sigma-n$. ◀

Note that, if k is in $O(c^n)$ for constant c, then $T(n,\sigma,k)\in O(n\log\sigma)$. Hence, in that case, our algorithm runs in $O(n\log\sigma)$ time. If $k\in O(1)$ our algorithm runs in optimal $O(n)$ time.

4.2 Deletions

▶ **Theorem 4.4.** *Let w be a word, with $|w|=n$, alph$(w)=\Sigma$, and $\Sigma=\{1,2,\ldots,\sigma\}$. Let k be an integer with $k\leq\iota(w)\leq n/\sigma$. We can compute in $O(nk)$ time the minimal number of deletions needed to obtain a word of universality index k (w.r.t. Σ) from w.*

The idea of this proof is the following. Assume that w' is a word of universality index k obtained via the sequence of deletions of minimal length from w. Clearly, w' is a subsequence of w, and, by the decomposition defined in Theorem 2.5, there exist $w'_1,\ldots,w'_k$, all of universality index exactly 1, and w'_{k+1}, of universality index 0, such that $w'=w'_1\cdots w'_kw'_{k+1}$. It follows that each of the words w'_i is a subsequence of w too. So we will try to identify each subsequence $w'_1\cdots w'_p$ for $p\leq k$ and the shortest factor $w[1:i]$ from which it is obtained. To this end, we define the matrix N, where $N[i][p]$ is the minimal number of deletions we need to apply to $w[1:i]$, without deleting $w[i]$, to obtain a word v from it, with $\iota(v)=p$

and $\iota(v[1:|v|-1]) = p-1$ (for $i \in [1:n]$ and $p \in [1:k]$). If $\iota(w[1:i]) \geq 1$, then $N[i][1] = |w[1:i]|_{w[i]} - 1$, as we have to delete all occurrences of $w[i]$ from $w[1:i]$, except the one on position i. Then, $N[i][p] = \min\{N[j][p-1] + |w[j+1:i]|_{w[i]} - 1 \mid j < i$ such that $\iota(w[j+1:i]) \geq 1\}$. This gives a dynamic programming algorithm for computing N. Using additional data structures extending the standard Range Minimum Queries structures (see Lemma 2.9), we can compute the elements of N in $O(nk)$ time. To show the statement, we return $\min\{N[i,k] + T[i] \mid 1 \leq i \leq n\}$, using the array T of Lemma 3.4.

4.3 Substitutions

▶ **Theorem 4.5.** *Let w be a word, with $|w| = n$, alph$(w) = \Sigma$, and $\Sigma = \{1,2,\ldots,\sigma\}$. Let k be an integer $0 \leq k \leq \lfloor \frac{n}{\sigma} \rfloor$. We can compute the minimal number of substitutions needed to apply to w in order to obtain a k-universal word (w.r.t. Σ) in $O(nk)$ time.*

The case $k > \iota(w)$ is treated similarly to the case of changing the universality of a word by insertions, described in Theorem 4.1. We define a matrix M, with $M[\ell][t]$ being the minimal number of substitutions one needs to apply to $w[1:\ell]$ in order to make it t-universal, for all $\ell \in [1:n]$ and all $t \in [1:k]$. Then, we derive the following recurrence: $M[\ell][t] = \min\{M[\ell'][t-1] + (\sigma - \Delta(\ell'+1,\ell)) \mid \ell'+1 \in \mathfrak{S}_\ell\}$, where $\mathfrak{S}_\ell = (S_\ell \cap [(t-1)\sigma : \ell-\sigma]) \cup \{\ell-\sigma+1\}$ (S_ℓ is defined in Section 3). The fact that at most σ elements of $M[\cdot][t-1]$ are used to compute each of the elements $M[\ell][t]$ allows us to apply Lemma 3.5 in almost the same way as we did in the algorithm of Theorem 4.1, and compute all the elements of M in $O(nk)$ time. The number we want to compute is $M[n][k]$.

For the case $k < \iota(w)$, we show that when decreasing the universality index of a word, it makes no difference whether we use substitutions or deletions. So, it is enough to use the algorithm of Theorem 4.4 and return the computed result as the answer to our current problem.

Note that, while substitutions and deletions can be used similarly to decrease the universality index of a word, we always need at least as many substitutions as insertions to increase it. To see that this inequality can also be strict, note that one insertion is enough to make `aabb` 2-universal, but we need two substitutions to achieve the same result.

5 Extensions

In this paper, we presented a series of algorithms computing the minimal number of edits one needs to apply to a word w in order to reach k-subsequence universality. In fact (see the proofs in the full version of this paper [19]), one can extend our algorithms and, using additional $O(k|\text{alph}(w)|)$ time, we can effectively construct a k-universal word which is closest to w, with respect to the edit distance. All our algorithms can be implemented in linear space (see [19]) using a technique called *Hirschberg's trick* [35].

The algorithms we presented work in a general setting: the processed words are over an integer alphabet. It seems natural to ask whether faster solutions for inputs over an alphabet of constant size (e.g., binary alphabets) exist. To this end, we state the following result.

▶ **Theorem 5.1.** *Let w be a word, with $|w| = n$, alph$(w) = \Sigma$, and $\Sigma = \{1,2,\ldots,\sigma\}$. Let k be an integer $\iota(w) < k$. We can compute the minimal number of insertions (resp., substitutions) needed to apply to w in order to obtain a k-universal word (w.r.t. Σ) in $(n \log^{O(1)} \sigma)$-time.*

We describe here the idea used in the case of insertions (as it works with small modifications for substitutions, too). Full details are given in [19]. We define a weighted directed acyclic graph G with the nodes $0, 1, \ldots, n$ and directed edges (i,j) with $i < j$. Let $\omega(i,j) = \sigma - \Delta(i+1,j)$ (i.e., the number of letter of Σ which do not appear in $w[i+1:j]$) be the weight of the edge (i, j). It is not hard to show that the number of edits needed to transform w into a k-universal word equals the weight of a minimum weight k-link path in G (see [7, 2, 58]). The graph G will not be explicitly constructed, but we can construct in $(n \log^{O(1)} \sigma)$-time an oracle data structure allowing us to retrieve in $O(\log \sigma / \log \log \sigma)$ the weight of any edge (i, j) of the graph. Further, as G fulfills the concave Monge property (i.e., $\omega(i, j) + \omega(i+1, j+1) < \omega(i+1, j) + \omega(i, j+1)$ holds for all $0 < i+1 < j < n$), then the minimum weight k-link path in G can be computed in $(n \log^{O(1)} \sigma)$-time, using the algorithms of [7, 2]. If $\sigma \in O(1)$, then the algorithms of Theorem 5.1 run in optimal linear time $O(n)$.

It is open whether a similar result holds for the case of decreasing the universality index of a word. However, the main open questions remaining from this work go back to our initial motivation. Namely, we would be very interested in settling the complexity of the following problem: Given a word w and a k-piecewise testable language L, succinctly specified, compute efficiently the edit distance from w to L. In particular, the case when L is defined as the language of words $\sim_k$-equivalent to a target-word u seems very interesting to us.

References

1 Alok Aggarwal, Maria M. Klawe, Shlomo Moran, Peter W. Shor, and Robert E. Wilber. Geometric applications of a matrix-searching algorithm. *Algorithmica*, 2:195–208, 1987.

2 Alok Aggarwal, Baruch Schieber, and Takeshi Tokuyama. Finding a minimum-weight k-link path graphs with the concae monge property and applications. *Discret. Comput. Geom.*, 12:263–280, 1994.

3 Cyril Allauzen and Mehryar Mohri. Linear-space computation of the edit-distance between a string and a finite automaton. *CoRR*, abs/0904.4686, 2009. arXiv:0904.4686.

4 Lorraine A. K. Ayad, Carl Barton, and Solon P. Pissis. A faster and more accurate heuristic for cyclic edit distance computation. *Pattern Recognit. Lett.*, 88:81–87, 2017.

5 Ricardo A. Baeza-Yates. Searching subsequences. *Theor. Comput. Sci.*, 78(2):363–376, 1991.

6 Laura Barker, Pamela Fleischmann, Katharina Harwardt, Florin Manea, and Dirk Nowotka. Scattered factor-universality of words. In Natasa Jonoska and Dmytro Savchuk, editors, *Proc. DLT 2020*, volume 12086 of *Lecture Notes in Computer Science*, pages 14–28, 2020.

7 Wolfgang W. Bein, Lawrence L. Larmore, and James K. Park. The d-edge shortest-path problem for a monge graph. *Technical Report*, SAND–92-1724C:1–10, 1992.

8 Michael A. Bender and Martin Farach-Colton. The LCA problem revisited. In *Proc. LATIN 2000*, volume 1776 of *Lecture Notes in Computer Science*, pages 88–94, 2000.

9 Giulia Bernardini, Huiping Chen, Grigorios Loukides, Nadia Pisanti, Solon P. Pissis, Leen Stougie, and Michelle Sweering. String sanitization under edit distance. In Inge Li Gørtz and Oren Weimann, editors, *Proc. CPM 2020*, volume 161 of *LIPIcs*, pages 7:1–7:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

10 Karl Bringmann and Bhaskar Ray Chaudhury. Sketching, streaming, and fine-grained complexity of (weighted) LCS. In *Proc. FSTTCS 2018*, volume 122 of *LIPIcs*, pages 40:1–40:16, 2018.

11 Karl Bringmann, Fabrizio Grandoni, Barna Saha, and Virginia Vassilevska Williams. Truly sub-cubic algorithms for language edit distance and RNA-folding via fast bounded-difference min-plus product. In *Proc. FOCS 2016*, pages 375–384, 2016.

12 Karl Bringmann and Marvin Künnemann. Multivariate fine-grained complexity of longest common subsequence. In *Proc. SODA 2018*, pages 1216–1235, 2018.

13 Krishnendu Chatterjee, Thomas A. Henzinger, Rasmus Ibsen-Jensen, and Jan Otop. Edit distance for pushdown automata. In *Proc. ICALP 2015*, volume 9135 of *Lecture Notes in Computer Science*, pages 121–133. Springer, 2015.

14 Herman Z. Q. Chen, Sergey Kitaev, Torsten Mütze, and Brian Y. Sun. On universal partial words. *Electronic Notes in Discrete Mathematics*, 61:231–237, 2017.

15 Hyunjoon Cheon and Yo-Sub Han. Computing the shortest string and the edit-distance for parsing expression languages. In *Proc. DLT 2020*, volume 12086 of *Lecture Notes in Computer Science*, pages 43–54, 2020.

16 Hyunjoon Cheon, Yo-Sub Han, Sang-Ki Ko, and Kai Salomaa. The relative edit-distance between two input-driven languages. In *Proc. DLT 2019*, volume 11647 of *Lecture Notes in Computer Science*, pages 127–139, 2019.

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

18 Maxime Crochemore, Borivoj Melichar, and Zdenek Tronícek. Directed acyclic subsequence graph - overview. *J. Discrete Algorithms*, 1(3-4):255–280, 2003.

19 Joel D. Day, Pamela Fleischmann, Maria Kosche, Tore Koß, Florin Manea, and Stefan Siemer. The edit distance to k-subsequence universality. *CoRR*, abs/2007.09192, 2020. arXiv:2007.09192.

20 Joel D. Day, Pamela Fleischmann, Florin Manea, and Dirk Nowotka. k-spectra of weakly-c-balanced words. In *Proc. DLT 2019*, volume 11647 of *Lecture Notes in Computer Science*, pages 265–277, 2019.

21 Nicolaas G. de Bruijn. A combinatorial problem. *Koninklijke Nederlandse Akademie v. Wetenschappen*, 49:758–764, 1946.

22 Aldo de Luca, Amy Glen, and Luca Q. Zamboni. Rich, sturmian, and trapezoidal words. *Theor. Comput. Sci.*, 407(1-3):569–573, 2008.

23 Xavier Droubay, Jacques Justin, and Giuseppe Pirillo. Episturmian words and some constructions of de Luca and Rauzy. *Theor. Comput. Sci.*, 255(1-2):539–553, 2001.

24 Lukas Fleischer and Manfred Kufleitner. Testing Simon's congruence. In *Proc. MFCS 2018*, volume 117 of *LIPIcs*, pages 62:1–62:13, 2018.

25 Dominik D. Freydenberger, Pawel Gawrychowski, Juhani Karhumäki, Florin Manea, and Wojciech Rytter. Testing k-binomial equivalence. In Multidisciplinary Creativity*, a collection of papers dedicated to G. Păun 65th birthday*, pages 239–248, 2015. available in CoRR abs/1509.00622.

26 Harold N. Gabow and Robert Endre Tarjan. A linear-time algorithm for a special case of disjoint set union. In *Proc. 15th STOC*, pages 246–251, 1983.

27 Emmanuelle Garel. Minimal separators of two words. In *Proc. CPM 1993*, volume 684 of *Lecture Notes in Computer Science*, pages 35–53, 1993.

28 Pawel Gawrychowski, Maria Kosche, Tore Koss, Florin Manea, and Stefan Siemer. Efficiently testing Simon's congruence. *CoRR*, abs/2005.01112, 2020. arXiv:2005.01112.

29 Pawel Gawrychowski, Martin Lange, Narad Rampersad, Jeffrey O. Shallit, and Marek Szykula. Existential length universality. In *Proc. STACS 2020*, volume 154 of *LIPIcs*, pages 16:1–16:14, 2020.

30 Bennet Goeckner, Corbin Groothuis, Cyrus Hettle, Brian Kell, Pamela Kirkpatrick, Rachel Kirsch, and Ryan W. Solava. Universal partial words over non-binary alphabets. *Theor. Comput. Sci*, 713:56–65, 2018.

31 Simon Halfon, Philippe Schnoebelen, and Georg Zetzsche. Decidability, complexity, and expressiveness of first-order logic over the subword ordering. In *Proc. LICS 2017*, pages 1–12, 2017.

32 R. W. Hamming. Error detecting and error correcting codes. *Bell Syst. Tech. J.*, 29(2):147–160, 1950.

33 Yo-Sub Han, Sang-Ki Ko, and Kai Salomaa. The edit-distance between a regular language and a context-free language. *Int. J. Found. Comput. Sci.*, 24(7):1067–1082, 2013.

34 Jean-Jacques Hebrard. An algorithm for distinguishing efficiently bit-strings by their subsequences. *Theor. Comput. Sci.*, 82(1):35–49, 22 May 1991.

35 Daniel S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Commun. ACM*, 18(6):341–343, 1975.

36 Markus Holzer and Martin Kutrib. Descriptional and computational complexity of finite automata - A survey. *Inf. Comput.*, 209(3):456–470, 2011.

37 Hiroshi Imai and Takao Asano. Dynamic segment intersection search with applications. In *Proc. FOCS 1984*, pages 393–402, 1984.

38 Rajesh Jayaram and Barna Saha. Approximating language edit distance beyond fast matrix multiplication: Ultralinear grammars are where parsing becomes hard! In *Proc. ICALP 2017*, volume 80 of *LIPIcs*, pages 19:1–19:15, 2017.

39 Prateek Karandikar, Manfred Kufleitner, and Philippe Schnoebelen. On the index of Simon's congruence for piecewise testability. *Inf. Process. Lett.*, 115(4):515–519, 2015.

40 Prateek Karandikar and Philippe Schnoebelen. The height of piecewise-testable languages with applications in logical complexity. In *Proc. CSL 2016*, volume 62 of *LIPIcs*, pages 37:1–37:22, 2016.

41 Prateek Karandikar and Philippe Schnoebelen. The height of piecewise-testable languages and the complexity of the logic of subwords. *Log. Methods Comput. Sci.*, 15(2), 2019.

42 Markus Krötzsch, Tomás Masopust, and Michaël Thomazo. Complexity of universality and related problems for partially ordered NFAs. *Inf. Comput.*, 255:177–192, 2017.

43 Dietrich Kuske. The subtrace order and counting first-order logic. In *Proc. CSR 2020*, volume 12159 of *Lecture Notes in Computer Science*, pages 289–302, 2020.

44 Dietrich Kuske and Georg Zetzsche. Languages ordered by the subword order. In *Proc. FOSSACS 2019*, volume 11425 of *Lecture Notes in Computer Science*, pages 348–364, 2019.

45 Marie Lejeune, Julien Leroy, and Michel Rigo. Computing the k-binomial complexity of the Thue-Morse word. In *Proc. DLT 2019*, volume 11647 of *Lecture Notes in Computer Science*, pages 278–291, 2019.

46 Julien Leroy, Michel Rigo, and Manon Stipulanti. Generalized Pascal triangle for binomial coefficients of words. *Electron. J. Combin.*, 24(1.44):36 pp., 2017.

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

48 Monroe H. Martin. A problem in arrangements. *Bull. Amer. Math. Soc.*, 40(12):859–864, December 1934.

49 Alexandru Mateescu, Arto Salomaa, and Sheng Yu. Subword histories and Parikh matrices. *J. Comput. Syst. Sci.*, 68(1):1–21, 2004.

50 Giovanni Pighizzini. How hard is computing the edit distance? *Inf. Comput.*, 165(1):1–13, 2001.

51 Jean-Eric Pin. The consequences of Imre Simon's work in the theory of automata, languages, and semigroups. In *Proc. LATIN 2004*, volume 2976 of *Lecture Notes in Computer Science*, page 5, 2004.

52 Jean-Eric Pin. The influence of Imre Simon's work in the theory of automata, languages and semigroups. *Semigroup Forum*, 98:1–8, 2019.

53 Narad Rampersad, Jeffrey Shallit, and Zhi Xu. The computational complexity of universality problems for prefixes, suffixes, factors, and subwords of regular languages. *Fundam. Inf.*, 116(1-4):223–236, January 2012.

54 Michel Rigo and Pavel Salimov. Another generalization of abelian equivalence: Binomial complexity of infinite words. *Theor. Comput. Sci.*, 601:47–57, 2015.

55 Jacques Sakarovitch and Imre Simon. Subwords. In M. Lothaire, editor, *Combinatorics on Words*, chapter 6, pages 105–142. Cambridge University Press, 1997.

56 Arto Salomaa. Connections between subwords and certain matrix mappings. *Theoret. Comput. Sci.*, 340(2):188–203, 2005.

57 David Sankoff and Joseph Kruskal. *Time Warps, String Edits, and Macromolecules The Theory and Practice of Sequence Comparison.* Cambridge University Press, 2000 (reprinted). originally published in 1983.

58 Baruch Schieber. Computing a minimum-weight k-link path in graphs with the concave monge property. In *Proc. SODA 1995*, pages 405–411. ACM/SIAM, 1995.

59 Shinnosuke Seki. Absoluteness of subword inequality is undecidable. *Theor. Comput. Sci.*, 418:116–120, 2012. doi:10.1016/j.tcs.2011.10.017.

60 Imre Simon. *Hierarchies of events with dot-depth one - Ph.D. thesis*. University of Waterloo, 1972.

61 Imre Simon. Piecewise testable events. In *Autom. Theor. Form. Lang., 2nd GI Conf.*, volume 33 of *LNCS*, pages 214–222, 1975.

62 Imre Simon. Words distinguished by their subwords (extended abstract). In *Proc. WORDS 2003*, volume 27 of *TUCS General Publication*, pages 6–13, 2003.

63 Zdenek Tronícek. Common subsequence automaton. In *Proc. CIAA 2002 (Revised Papers)*, volume 2608 of *Lecture Notes in Computer Science*, pages 270–275, 2002.

64 Robert A. Wagner. Order-n correction for regular languages. *Commun. ACM*, 17(5):265–268, 1974.

65 Robert A. Wagner and Michael J. Fischer. The string-to-string correction problem. *J. ACM*, 21(1):168–173, January 1974.

66 Georg Zetzsche. The complexity of downward closure comparisons. In *Proc. ICALP 2016*, volume 55 of *LIPIcs*, pages 123:1–123:14, 2016.