Fundamenta Informaticae 189(3-4) : 199–240 (2022)
Available at IOS Press through:
https://doi.org/10.3233/FI-222159

arXiv:2108.13968v6 [cs.FL] 11 Oct 2023

Absent Subsequences in Words

Maria Kosche, Tore Koß*, **Florin Manea, Stefan Siemer**
Institute for Computer Science
Georg-August University Göttingen, Germany
{*maria.kosche,tore.koss,florin.manea,stefan.siemer*}@*cs.uni-goettingen.de*

Abstract. An absent factor of a string w is a string u which does not occur as a contiguous substring (a.k.a. factor) inside w. We extend this well-studied notion and define absent subsequences: a string u is an absent subsequence of a string w if u does not occur as subsequence (a.k.a. scattered factor) inside w. Of particular interest to us are minimal absent subsequences, i.e., absent subsequences whose every subsequence is not absent, and shortest absent subsequences, i.e., absent subsequences of minimal length. We show a series of combinatorial and algorithmic results regarding these two notions. For instance: we give combinatorial characterisations of the sets of minimal and, respectively, shortest absent subsequences in a word, as well as compact representations of these sets; we show how we can test efficiently if a string is a shortest or minimal absent subsequence in a word, and we give efficient algorithms computing the lexicographically smallest absent subsequence of each kind; also, we show how a data structure for answering shortest absent subsequence-queries for the factors of a given string can be efficiently computed.

Keywords: Absent subsequence Arch-factorization Stringology Subsequence Subsequence- Universality.

1. Introduction

A word u is a subsequence (also called scattered factor or subword) of a string w if there exist (possibly empty) strings $v_1, \ldots, v_{\ell+1}$ and $u_1, \ldots, u_\ell$ such that $u = u_1 \ldots u_\ell$ and $w = v_1 u_1 \ldots v_\ell u_\ell v_{\ell+1}$. In other words, u can be obtained from w by removing some of its letters.

*Address for correspondence: Institute for Computer Science, Georg-August University Göttingen, Germany

The study of the relationship between words and their subsequences has been a central topic in combinatorics on words and string algorithms, as well as in language and automata theory (see, e.g., the chapter *Subwords* in [1, Chapter 6] for an overview of the fundamental aspects of this topic). Subsequences appear in many areas of theoretical computer science, such as logic of automata theory [2, 3, 4, 5, 6, 7, 8, 9, 10], combinatorics on words [11, 12, 13, 14, 15, 16, 17], as well as algorithms [18, 19, 20, 21, 22]. From a practical point of view, subsequences are generally used to model corrupted or lossy representations of an original string, and appear, for instance, in applications related to formal verification, see [2, 10] and the references therein, or in bioinformatics-related problems, see [23].

In most investigations related to subsequences, comparing the sets of subsequences of two different strings is usually a central task. In particular, Imre Simon defined and studied (see [1, 8, 9]) the relation $\sim_k$ (now called the Simon's Congruence) between strings having exactly the same set of subsequences of length at most k (see, e.g., [5] as well as the surveys [24, 25] and the references therein for the theory developed around $\sim_k$ and its applications). In particular, $\sim_k$ is a well-studied relation in the area of string algorithms. The problems of deciding whether two given strings are $\sim_k$-equivalent, for a given k, and to find the largest k such that two given strings are $\sim_k$-equivalent (and their applications) were heavily investigated in the literature, see, e.g., [26, 27, 28, 29, 30, 31] and the references therein. Last year, optimal solutions were given for both these problems [32, 33]. Two concepts seemed to play an important role in all these investigations: on the one hand, the notion of distinguishing word, i.e., the shortest subsequence present in one string and *absent* from the other. On the other hand, the notion of universality index of a string [32, 34], i.e., the largest k such that the string contains as subsequences all possible strings of length at most k; that is, the length of the shortest subsequence *absent* from that string, minus 1.

Motivated by these two concepts and the role they play, we study in this paper the set of *absent subsequences* of a string w, i.e., the set of strings which *are not* subsequences of w. As such, our investigation is also strongly related to the study of *missing factors* (or missing words, MAWs) in strings, where the focus is on the set of strings which are not substrings (or factors) of w. The literature on the respective topic ranges from many very practical applications of this concept [35, 36, 37, 38, 39, 40] to deep theoretical results of combinatorial [41, 42, 43, 44, 45, 46, 47] or algorithmic nature [48, 49, 35, 50, 51, 52, 53]. Absent subsequences are also related to the well-studied notion of patterns avoided by permutations, see for instance [54], with the main difference being that a permutation is essentially a word whose letters are pairwise distinct.

Moreover, absent subsequences of a string (denoted by w in the following) seem to naturally occur in many practical scenarios, potentially relevant in the context of reachability and avoidability problems.

On the one hand, assume that w is some string (or stream) we observe, which may represent e.g. the trace of some computation or, in a totally different framework, the DNA-sequence describing some gene. In this framework, absent subsequences correspond to sequences of letters avoided by the string w. As such, they can be an avoided sequence of events in the observed trace or an avoided scattered sequence of nucleotides in the given gene. Understanding the set of absent subsequences of the respective string, and in particular its extremal elements with respect to some ordering, as well as being able to quickly retrieve its elements and process them efficiently seems useful to us.

On the other hand, when considering problems whose input is a set of strings, one could be interested in the case when the respective input can be compactly represented as the set of absent subsequences (potentially with some additional combinatorial properties, which make this set finite) of a given string. Clearly, one would then be interested in processing the given string and representing its set of absent subsequences by some compact data structure which further would allow querying it efficiently.

In this context, our paper is focused on two particular classes of absent subsequences: *minimal absent subsequences* (MAS for short), i.e., absent subsequences whose every subsequence is not absent, and *shortest absent subsequences* (SAS for short), i.e., absent subsequences of minimal length. In Section 3, we show a series of novel combinatorial results: we give precise characterizations of the set of minimal absent subsequences and shortest absent subsequences occurring in a word, as well as examples of words w having an exponential number (w.r.t. the length of w) of minimal absent subsequences and shortest absent subsequences, respectively. We also identify, for a given number k, a class of words having a maximal number of SAS, among all words whose SAS have length k.

We continue with a series of algorithmic results in Section 4. We first show a series of simple algorithms, useful to test efficiently if a string is a shortest or minimal absent subsequence in a word. Motivated by the existence of words with exponentially large sets of minimal absent subsequences and shortest absent subsequences, our main contributions show, in Sections 4.1 and 4.2, how to construct compact representations of these sets. These representations are fundamental to obtaining efficient algorithms querying the set of SAS and MAS of a word, and searching for such absent subsequences with certain properties or efficiently enumerating them. These results are based on the combinatorial characterizations of the respective sets combined with an involved machinery of data structures, which we introduce gradually for the sake of readability. In Section 4.1, we show another main result of our paper, where, for a given word w, we construct in linear time a data structure for efficiently answering queries asking for the shortest absent subsequences in the factors of w (note that the same problem was recently approached in the case of missing factors [49]).

The techniques used to obtain these results are a combination of combinatorics on words results with efficient data structures and algorithmic techniques.

2. Basic definitions

Let $\mathbb{N}$ be the set of natural numbers, including 0. For $m, n \in \mathbb{N}$, we let $[m : n] = \{m, m+1, \dots, n\}$. An alphabet Σ is a nonempty finite set of symbols called *letters*. A *string (also called word)* is a finite sequence of letters from Σ, thus an element of the free monoid Σ^*. For the rest of the paper, we assume that the strings we work with are over an alphabet $\Sigma = \{1, 2, \dots, \sigma\}$.

Let $\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$, where ε is the empty string. The *length* of a string $w \in \Sigma^*$ is denoted by $|w|$. The ith letter of $w \in \Sigma^*$ is denoted by $w[i]$, for $i \in [1 : |w|]$. For $m, n \in \mathbb{N}$, we let $w[m : n] = w[m]w[m+1] \dots w[n]$, $|w|_a = |\{i \in [1 : |w|] \mid w[i] = a\}|$. A string $u = w[m : n]$ is a *factor* of w, and we have $w = xuy$ for some $x, y \in \Sigma^*$. If $x = \varepsilon$ (resp. $y = \varepsilon$), u is called a *prefix* (resp. *suffix*) of w. Let $\mathrm{alph}(w) = \{x \in \Sigma \mid |w|_x > 0\}$ be the smallest subset $S \subset \Sigma$ such that $w \in S^*$. We can now introduce the notion of subsequence.

Definition 2.1. We call u a subsequence of length k of w, where $|w| = n$, if there exist positions $1 \leq i_1 < i_2 < \ldots < i_k \leq n$, such that $u = w[i_1]w[i_2] \cdots w[i_k]$.

We recall the notion of k-universality of a string as presented in [32].

Definition 2.2. We call a word w k-universal if any string v of length $|v| \leq k$ over $\mathrm{alph}(w)$ appears as a subsequence of w. For a word w, we define its universality index $\iota(w)$ to be the largest integer k such that w is k-universal.

If $\iota(w) = k$, then w is ℓ-universal for all $\ell \leq k$. Note that the universality index of a word w is always defined w.r.t. the alphabet of the word w. For instance, $w = 01210$ is 1-universal (as it contains all words of length 1 over $\{0, 1, 2\}$ but would not be 1-universal if we consider an extended alphabet $\{0, 1, 2, 3\}$. The fact that the universality index is computed w.r.t. the alphabet of w also means that every word is at least 1-universal. Note that in our results we either investigate the properties of a given word w or we show algorithms working on some input word w. In this context, the universality of the factors of w and other words we construct is defined w.r.t. $\mathrm{alph}(w)$. See [32, 34] for a detailed discussion on this.

We recall the arch factorisation, introduced by Hebrard [28].

Definition 2.3. For $w \in \Sigma^*$, with $\Sigma = \mathrm{alph}(w)$, the *arch factorisation* of w is defined as $w = \mathrm{ar}_w(1) \cdots \mathrm{ar}_w(\iota(w))\,\mathrm{r}(w)$ where for all $i \in [1 : \iota(w)]$ the last letter of $\mathrm{ar}_w(i)$ occurs exactly once in $\mathrm{ar}_w(i)$, each arch $\mathrm{ar}_w(i)$ is 1-universal, and $alph(\mathrm{r}(w)) \subsetneq \Sigma$. The words $\mathrm{ar}_w(i)$ are called *arches* of w, $\mathrm{r}(w)$ is called the *rest*.

Let $m(w) = \mathrm{ar}_w(1)[|\mathrm{ar}_w(1)|] \cdots \mathrm{ar}_w(k)[|\mathrm{ar}_w(k)|]$ be the word containing the unique last letters of each arch.

Note that every word has a unique arch factorisation and by definition each arch $\mathrm{ar}_w(i)$ from a word w is 1-universal. By an abuse of notation, we can write $i \in \mathrm{ar}_w(\ell)$ if i is a natural number such that $|\mathrm{ar}_w(1) \cdots \mathrm{ar}_w(\ell - 1)| < i \leq |\mathrm{ar}_w(1) \cdots \mathrm{ar}_w(\ell)|$, i.e., i is a position of w contained in the ℓth arch of w.

The main concepts discussed in this paper are the following.

Definition 2.4. A word v is an absent subsequence of w if v is not a subsequence of w. An absent subsequence v of w is a minimal absent subsequence (for short, MAS) of w if every proper subsequence of v is a subsequence of w. We will denote the set of all MAS of w by $\mathrm{MAS}(w)$. An absent subsequence v of w is a shortest absent subsequence (for short, SAS) of w if $|v| \leq |v'|$ for any other absent subsequence v' of w. We will denote the set of all SAS of w by $\mathrm{SAS}(w)$.

Note that any SAS of w has length $\iota(w) + 1$ and v is an MAS of w if and only if v is absent and every subsequence of v of length $|v| - 1$ is a subsequence of w.

3. Combinatorial properties of SAS and MAS

We begin with a presentation of several combinatorial properties of the MAS and SAS. Let us first take a closer look at MAS.

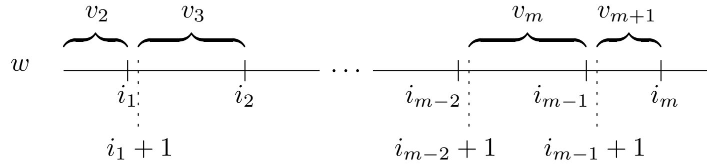

Figure 1. Illustration of positions and intervals inside word w

If $v = v[1] \cdots v[m+1]$ is an MAS of w then $v[1] \cdots v[m]$ is a subsequence of w. Hence, we can go left-to-right through w and greedily choose positions $1 \leq i_1 < \ldots < i_m \leq n = |w|$ such that $v[1] \cdots v[m] = w[i_1] \cdots w[i_m]$ and i_ℓ is the leftmost occurrence of $w[i_\ell]$ in $w[i_{\ell-1}+1 : n]$ (as described in Algorithm 4 in Section 4). Because v itself is absent, $v[m+1]$ cannot occur in the suffix of w starting at $i_m + 1$. Furthermore, we know that $v[1] \cdots v[m-1]v[m+1]$ is a subsequence of w. Hence, $v[m+1]$ occurs in the suffix of w starting at $i_{m-1}+1$. We deduce $v[m+1] \in \mathrm{alph}(w[i_{m-1}+1 : i_m]) \setminus \mathrm{alph}(w[i_m + 1 : n])$. This argument is illustrated in Figure 1 and can be applied inductively to deduce $v[k] \in \mathrm{alph}(w[i_{k-2} + 1 : i_{k-1}]) \setminus \mathrm{alph}(w[i_{k-1} + 1 : i_k - 1])$ for all $k \neq 1$. The choice of $v[1] \in \mathrm{alph}(w)$ is arbitrary. More details are given in the proof to the following theorem. For notational reasons we introduce $i_0 = 0$ and $i_{m+1} = n + 1$.

Theorem 3.1. Let $v, w \in \Sigma^*$, $|v| = m + 1$ and $|w| = n$, then v is an MAS of w if and only if there are positions $0 = i_0 < i_1 < \ldots < i_m < i_{m+1} = n + 1$ such that all of the following conditions are satisfied.

(i) $v = w[i_1] \cdots w[i_m]v[m+1]$

(ii) $v[1] \notin \mathrm{alph}(w[1 : i_1 - 1])$

(iii) $v[k] \notin \mathrm{alph}(w[i_{k-1} + 1 : i_k - 1])$ for all $k \in [2 : m+1]$

(iv) $v[k] \in \mathrm{alph}(w[i_{k-2} + 1 : i_{k-1}])$ for all $k \in [2 : m+1]$

Proof:
Let v be an MAS of w of length $|v| = m + 1$. By definition, $v[1 : m]$ is a subsequence of w, hence we can choose positions $i_1, \ldots i_m$ such that $v[\ell] = w[i_\ell]$, for all $\ell \in [1 : m]$. We do this greedily, that is, we choose i_1 to be the leftmost occurrence of $v[1]$ in w, i_2 be the leftmost occurrence of $v[2]$ in $w[i_1+1 : n]$ and so on. In the end, $v[m+1]$ cannot occur in $w[i_m+1 : n]$, because v is no subsequence of w. Now, the positions $i_1, \ldots, i_m$ and $i_{m+1} = n + 1$ clearly satisfy conditions (i) to (iii).

We show first that $k = m+1$ satisfies condition (iv): $v[1 : m-1]v[m+1]$ is a subsequence of w, hence $v[m+1]$ occurs in $w[i_{m-1}+1 : n]$. By condition (iii) $v[m+1]$ does not occur in $w[i_m+1 : n]$ hence condition (iv) is true for $k = m + 1$. Now let $2 \leq \ell < m + 1$, we make two observations:

1. $v[\ell : m+1]$ is absent in $w[i_{\ell-1} + 1 : n]$ because v is absent in w,
2. $v[\ell + 1 : m + 1]$ is a subsequence of $w[i_{\ell-1} + 1 : n]$ because v is a MAS of w.

Combining both observations yields that $v[\ell : m+1]$ is a subsequence of $w[i_{\ell-2}+1 : n]$ if and only if $v[\ell]$ occurs in $w[i_{\ell-2}+1 : i_{\ell-1}]$. Since v is an MAS of w, $v[1 : \ell-2]v[\ell : m+1]$ is a subsequence of w, hence $v[\ell : m+1]$ is a subsequence of $w[i_{\ell-2}+1 : n]$ and so $v[\ell] \in \mathrm{alph}(w[i_{\ell-2}+1 : i_{\ell-1}])$.

Now let $i_0 = 0$, $i_{m+1} = n+1$ and $i_1, \ldots, i_m$ be positions of w satisfying conditions (i) to (iv). From condition (i), we have that $v = w[i_1] \cdots w[i_m]v[m+1]$. We claim that v is an MAS of w. Firstly, we divide w into $w = w_1 \cdots w_m \, \mathrm{r}(w)$ where $w_k = w[i_{k-1}+1 : i_k]$ and $\mathrm{r}(w) = w[i_m+1 : n]$. By condition (iii), $v[m+1]$ doesn't occur in $\mathrm{r}(w)$, hence v is absent in w.

For the minimality, we notice that by condition (iv), $v[k]$ occurs in w_{k-1}. Therefore, if we choose an arbitrary $\ell \leq m+1$ and delete $v[\ell]$, we have $v[1 : \ell-1] = w[i_1] \cdots w[i_{\ell-1}]$ is a subsequence of $w_1 \cdots w_{\ell-1}$, and $v[\ell+1 : m+1]$ is a subsequence of $w_\ell \cdots w_m$, hence $v[1 : \ell-1]v[\ell+1 : m+1]$ is a subsequence of w. The conclusion follows. □

Properties (i) to (iii) (the latter for $k \leq m$ only) are satisfied if we choose the positions $i_1, \ldots, i_m$ greedily, as described in the beginning of this section.

By Theorem 3.1, we have no restriction on the first letter of an MAS and indeed we can find an MAS starting with an arbitrary letter.

Remark 3.2. For every $x \in \mathrm{alph}(w)$, $x^{|w|_x+1}$ is an MAS of w, hence, for every choice of $x \in \mathrm{alph}(w)$, we can find an MAS v starting with x.

Using Theorem 3.1 we can now determine the whole set of MAS of a word w. This will be formalized later in Theorem 4.21. For now, we just give an example.

Example 3.3. Let $w = 0011 \in \{0,1\}^*$ and we want to construct v, an MAS of w. We start by choosing $v[1] = 0$. Then $i_1 = 1$ and by item (iv) $v[2] \in \mathrm{alph}(w[1 : 1]) = \{0\}$, so $i_2 = 2$ (by item (iii)). Again by item (iv), we have $v[3] \in \mathrm{alph}(w[2 : 2]) = \{0\}$. The letter $v[3]$ does not occur in $w[i_2+1 : n]$, hence, $i_3 = n+1$, and $v = 0^3$ is an MAS of w. If we let $v[1] = 1$, we have $i_1 = 3$. By item (iv), we have $v[2] \in \mathrm{alph}(w[1 : 3]) = \{0,1\}$. If we choose $v[2] = 1$, we obtain $v = 1^3$ with an argument analogous to the first case. So let us choose $v[2] = 0$. Then $i_2 = n+1$, and $v = 10$ is an MAS of w. Theorem 3.1 claims $\mathrm{MAS}(w) = \{0^3, 10, 1^3\}$ and indeed 10 is the only absent sequence of length 2, and every word of length ≥ 3 is either not absent (001, 011 and 0011) or contains 10, 0^3 or 1^3 as a subsequence.

From this example also follows that not every MAS is an SAS. The converse is necessarily true. So, for any SAS v of w we have $|v| = \iota(w)+1$, and we can find positions $1 \leq i_1 < i_2 < \ldots < i_{\iota(w)} \leq n$ satisfying Theorem 3.1. The following theorem claims that every arch of w (see Definition 2.3) contains exactly one of these positions.

Theorem 3.4. Let $w = \mathrm{ar}_w(1) \cdots \mathrm{ar}_w(\iota(w)) \, \mathrm{r}(w)$ as in Definition 2.3. Then, v is an SAS of w if and only if there are positions $i_0 = 0$, $i_\ell \in \mathrm{ar}_w(\ell)$ for all $1 \leq \ell \leq \iota(w)$, and $i_{\iota(w)+1} = n+1$ satisfying Theorem 3.1.

Proof:
Let v be an SAS of w. Every SAS is an MAS, so we can choose positions $i_1, \ldots i_{\iota(w)}$ as in Theorem 3.1. We claim that every arch of w contains exactly one of these positions. Because the number of these positions equals the number of arches, it suffices to show that every arch contains at least one i_ℓ. Assume there is an arch $\mathrm{ar}_w(k)$ not containing any of the is. Because, by conditions (ii) and (iii) of Theorem 3.1, we choose positions greedily and every arch is 1-universal this is only possible if v is a subsequence of $\mathrm{ar}_w(1) \cdots \mathrm{ar}_w(k-1)$. This is a contradiction because v is absent from w.

For the converse implication, we have a word $v = w[i_1] \cdots w[i_{\iota(w)}]v[\iota(w)+1]$ which satisfies Theorem 3.1. Hence v is an MAS of length $|v| = \iota(w)+1$ and thus an SAS. □

A way to efficiently enumerate all SAS in a word will be given later in Theorem 4.19. Here, we only give an example based on a less efficient, but more intuitive, strategy of identifying the SAS of a word.

Example 3.5. Let $w = 012121012$ with $\iota(w) = 2$, and the arch factorisation of w is $w = 012 \cdot 1210 \cdot 12$. We construct v, an SAS of w. By Theorem 3.4, we have $|v| = \iota(w)+1 = 3$ and by condition (iv) of Theorem 3.1 the letter $v[3]$ does not occur in $\mathrm{alph}(w[i_2 : n]) \supset \mathrm{alph}(\mathrm{r}(w))$, so $v[3]$ is not contained in $\mathrm{alph}(r(w))$. Hence, $v[3] = 0$ and its rightmost position in $\mathrm{ar}_w(2)$ is on position 7. Therefore, $v[2]$ should not appear before position 7 in $\mathrm{ar}_w(2)$ (as $v[1]$ appears in $\mathrm{ar}_w(1)$ for sure). So, $v[2] \notin \mathrm{alph}(w[4:6]) = \{1,2\}$ and $v[2] = 0$. Ultimately, the rightmost occurrence of $v[2]$ in $\mathrm{ar}_w(1)$ is on position 1, and we can arbitrarily choose $v[1] \in \mathrm{alph}(\mathrm{ar}_w(1)) = \{0,1,2\}$. We conclude that $\mathrm{SAS}(w) = \{000, 100, 200\}$.

To better understand the properties of SAS and MAS, we analyse some particular words. For the rest of this section, assume that $\sigma \geq 4$ is a large even natural constant.

Firstly we let $v_1 = 1 \cdot 2 \cdots \sigma$ and $v_2 = \sigma \cdot \sigma - 1 \cdots 1$. Then, for $k \in \mathbb{N}$, we define the words $A_{2k} = (v_1 \cdot v_2)^k$, $A_{2k+1} = A_{2k} \cdot v_1$, and $B_k = v_1^k$. Each of these words has universality index $\iota(A_k) = \iota(B_k) = k$ and every arch has minimal length $|\mathrm{ar}_{A_k}(i)| = |\mathrm{ar}_{B_k}(i)| = |\Sigma|$ for all $k \in \mathbb{N}$ and $i \in [1 : k]$. Furthermore the SAS of B_k are exactly the monotonically decreasing sequences of length $k+1$, whereas no SAS of A_k has a strictly monotonically increasing or decreasing factor of length 3. We will devote the rest of this chapter to an analysis of the sets $\mathrm{SAS}(w)$ and $\mathrm{MAS}(w)$ for w being either A_k or B_k. It turns out that B_k has a small (that is polynomial in $|B_k|$) amount of SAS but the amount of MAS is exponential (in $|B_k|$) whereas every MAS of A_k is an SAS and A_k has the maximal amount of SAS among all words w with universality index $\iota(w) = k$. We start with the following result.

Proposition 3.6. The word B_k has a polynomial number of SAS and exponentially (in the length of the word) more MAS than SAS.

Proof:
Recall our assumption that $\sigma \geq 4$ is a large even constant, and let $k = (2\sigma+2)m$ where m is a natural number. Let v be an SAS of B_k. From Theorem 3.4, it follows that $v[i] \geq v[i+1]$, for all $i \in [1 : |v|]$. So, v is a decreasing sequence of numbers from $\{1, \ldots, \sigma\}$, of length $k+1$. Let us count these sequences. We first count the number of sequences whose elements are contained in a

fixed set $T \subset \{1, \ldots, \sigma\}$ of size $|T| = t$. For each T there are exactly $\binom{k}{t-1}$ such sequences (as we essentially need to choose, for each value except the smallest one, of the t values, the rightmost element of the sequence which takes that certain value and $k+1$ is necessarily reserved for the smallest value). Moreover, we can choose the t values taken by the numbers in the sequence in $\binom{\sigma}{t}$ ways. So, the number of decreasing sequences of numbers from $\{1, \ldots, \sigma\}$, of length $k+1$, is $S_{k+1} = \sum_{t=1}^{\sigma} \left(\binom{\sigma}{t} \cdot \binom{k}{t-1} \right)$. Since $t - 1 < \sigma \leq \frac{k}{2}$ we have $\binom{k}{t-1} \leq \binom{k}{t}$ and therefore

$$S_{k+1} \leq \sum_{t=1}^{\sigma} \left(\binom{\sigma}{t} \cdot \binom{k}{t} \right) \leq \sum_{t=1}^{\sigma} \left(\left(\frac{e\sigma}{t} \right)^t \cdot \left(\frac{ek}{t} \right)^t \right) = \sum_{t=1}^{\sigma} \frac{(e^2 \sigma k)^t}{t^{2t}}.$$

Applying Chebyshev's sum inequality we obtain that the following holds:

$$S_{k+1} \leq \frac{1}{\sigma} \cdot \left(\sum_{t=1}^{\sigma} (e^2 \sigma k)^t \right) \cdot \left(\sum_{t=1}^{\sigma} \frac{1}{t^{2t}} \right).$$

Thus,

$$S_{k+1} \leq \frac{1}{\sigma} \cdot \frac{(e^2 \sigma k)^{\sigma+1} - 1}{(e^2 \sigma k) - 1} \cdot \frac{\pi^2}{6} \leq 4 \cdot \frac{(e^2 \sigma k)^{\sigma}}{\sigma}.$$

As σ is assumed to be constant, it follows that S_{k+1} is bounded by a polynomial from above. We can also prove a lower bound for S_{k+1}. We have that:

$$\begin{aligned} S_{k+1} &\geq \sum_{t=1}^{\sigma} \left(\left(\frac{\sigma}{t} \right)^t \cdot \left(\frac{k}{t-1} \right)^{t-1} \right) \geq \sum_{t=1}^{\sigma} \left(\left(\frac{\sigma}{t} \right)^{t-1} \cdot \left(\frac{k}{t} \right)^{t-1} \right) \\ &= \sum_{t=1}^{\sigma} \left(\frac{(\sigma k)}{t^2} \right)^{t-1} \geq \sum_{t=1}^{\sigma} \left(\frac{(\sigma k)}{\sigma^2} \right)^{t-1}. \end{aligned}$$

Thus,

$$S_{k+1} \geq \frac{(k/\sigma)^{\sigma} - 1}{(k/\sigma) - 1} \geq (k/\sigma)^{\sigma - 1}.$$

A family of words from $\mathrm{MAS}(B_k)$, which are not in $\mathrm{SAS}(B_k)$, are the words

$$v = \left(\Pi_{r \in [1:m]} (\sigma \cdot u_r \cdot 1 \cdot \sigma) \right) \sigma,$$

where $\Pi_{r \in [1:m]}$ is used to denote concatenation of the terms and, for all r, u_r is a decreasing sequence of length 2σ of numbers from $\{1, \ldots, \sigma\}$ not ending in σ, that is u_i is not the constant sequence consisting in 2σ occurrences of the letter σ.

We will show that v is a MAS of $B_k = v_1^k$ by showing that it satisfies Theorem 3.1: since the u_r are decreasing and v_1 is strictly increasing, the only factor of length 2 of v which is a subsequence of v_1 is $1 \cdot \sigma$. Hence when searching for positions $i_1, i_2, \ldots i_{|v|-1}$ satisfying item (i) positions i_j and i_{j+1} can be found inside the same arch of B_k if and only if $v[j : j+1] = 1 \cdot \sigma$. So, for $1 \leq \ell \leq m$ the prefix $\Pi_{r \in [1:\ell]} (\sigma \cdot u_r \cdot 1 \cdot \sigma)$ of v is a subsequence of the prefix $v_1^{(|u_r|+2)\ell}$ of B_k but absent from any

shorter prefix of B_k. Since $B_k = v_1^k = v_1^{(2\sigma+2)m} = v_1^{(|u_r|+2)m}$, v is absent from B_k and its prefix of length $|v|-1$ is a subsequence of B_k. Furthermore, as every letter of Σ occurs exactly once in v_1, there is only one possibility to choose indices $i_1, i_2, \ldots, i_{|v|-1}$ satisfying item (i) of Theorem 3.1, which is, $i_1 = \sigma$ and consecutive positions i_j, i_{j+1} are uniquely chosen from consecutive arches of B_k unless $v[j] = 1$ and $v[j+1] = \sigma$ in which case i_j and i_{j+1} are uniquely chosen from the same arch. We will show that these indices satisfy items (ii) to (iv) of Theorem 3.1 as well. Firstly $v[1] = \sigma$ does not occur in $B_k[1 : i_1 - 1] = 1 \cdot 2 \cdots \sigma - 1$. Secondly recall that $v[j] > v[j-1]$ if and only if $v[j-1] = 1$ and $v[j] = \sigma$, thus

$$v[j] \notin \mathrm{alph}(B_k[i_{j-1}+1 : i_j - 1]) = \begin{cases} \Sigma \setminus [v[j] : v[j-1]], & \text{if } v[j] \leq v[j-1] \\ \{2, 3, \ldots, \sigma - 1\}, & \text{if } v[j] > v[j-1]. \end{cases}$$

For the last item of Theorem 3.1 note that if $v[j-1] \leq v[j-2]$ then either $v[j] \leq v[j-1]$ or $v[j] = \sigma > u_i[|u_i|] = v[j-2]$. Otherwise, if $v[j-1] > v[j-2]$, then $v[j] = \sigma$. Thus

$$v[j] \in \mathrm{alph}(B_k[i_{j-2}+1 : i_{j-1}]) = \begin{cases} \Sigma \setminus [v[j-1]+1 : v[j-2]], & \text{if } v[j-1] \leq v[j-2] \\ \{2, 3, \ldots, \sigma\}, & \text{if } v[j-1] > v[j-2]. \end{cases}$$

Based on the results shown above about SAS and the assumption that $\sigma \geq 4$, the size S' of the family of words v as defined above fulfils:

$$S' \geq (S_{2\sigma} - 1)^m \geq \left(\left(\frac{2\sigma - 1}{\sigma} \right)^{\sigma - 1} - 1 \right)^m \geq 4^m.$$

Given that m is not a constant, but a parameter which grows proportional to $|B_k|$, the statement of the theorem follows. □

Based on Theorems 3.1 and 3.4, it follows that the SAS of B_k correspond to decreasing sequences of length $k+1$ of numbers from $\{1, \ldots, \sigma\}$. A family of words included in $\mathrm{MAS}(B_k)$, which is disjoint from $\mathrm{SAS}(B_k)$, consists of the words $v = (\Pi_{t=1,m}(\sigma \cdot u_i \cdot 1 \cdot \sigma))\, \sigma$, where u_i is a decreasing sequence of length 2σ of numbers from $\{1, \ldots, \sigma\}$, for all i. By counting arguments, we obtain the result stated above. However, there are words whose sets of MAS and SAS coincide.

Proposition 3.7. $\mathrm{MAS}(A_k) = \mathrm{SAS}(A_k)$.

Proof:
Assume that there exist an MAS v of A_k which is not an SAS (i.e., $|v| = m$ is strictly greater than $k+1$). We go left-to-right through A_k and greedily choose positions $i_1 < \ldots < i_m \leq k\sigma$ such that $v[1] \cdots v[m] = A_k[i_1] \cdots A_k[i_m]$ and i_j is the leftmost occurrence of $A_k[i_j]$ in $A_k[i_{j-1}+1 : k\sigma]$. Let $a_j = A_k[i_j]$ for $j \in [1 : m]$. As $m - 1 > k$, there exists an arch $\mathrm{ar}_{A_k}(\ell)$ of A_k which contains at least two positions i_ℓ and $i_{\ell+1}$, and all archs $\mathrm{ar}_{A_k}(\ell')$ with $\ell' < \ell$ contain exactly one such position. We can assume without loss of generality that ℓ is odd (the even case is identical). If $\ell > 1$ then it is easy to see that $a_{\ell-1} \leq a_\ell$ (otherwise a_ℓ would be in the same arch as $a_{\ell-1}$) and $a_\ell < a_{\ell+1}$ (as $a_{\ell+1}$ is to the right of a_ℓ in an odd arch). So, $a_{\ell-1} < a_{\ell+1}$. It follows that when removing $v[\ell]$ from v, we obtain a word which is not a subsequence of A_k. If $\ell = 1$ we get the same conclusion, trivially. This is a contradiction to the fact that v is an MAS. So our assumption was wrong, and v must be an SAS. □

We can actually show a slightly stronger property: any word which has no proper SAS is a permutation of A_k for a suitable k. For the rest of this chapter π always denotes a morphism of Σ^* that acts as a permutation on the letters of Σ, that is, π is a bijection of Σ into itself and satisfies $\pi(uv) = \pi(u)\pi(v)$ for all $u, v \in \Sigma^*$. For simplicity we call π a permutation (of Σ). In contrast we denote any permutation of integers, e.g. positions of a word, by θ. For illustration let π and θ both denote the permutation that maps 1 to 2 and vice versa, then $\pi(1322) = 2311$ and $\theta(1322) = 3122$.

The first observation follows directly.

Lemma 3.8. If π is a permutation of Σ and $w \in \Sigma^*$ then $\pi(v)$ is an SAS of $\pi(w)$ if and only if v is an SAS of w. Hence $|\operatorname{SAS}(w)| = |\operatorname{SAS}(\pi(w))|$.

Proposition 3.9. Let $w \in \Sigma^*$, then $|\operatorname{SAS}(w)| = |\operatorname{MAS}(w)|$ if and only if there is a permutation π such that $\pi(w) = A_{\iota(w)}$.

Proof:
The if-part is immediate: $|\operatorname{SAS}(w)| = |\operatorname{SAS}(\pi(w))| = |\operatorname{MAS}(\pi(w))| = |\operatorname{MAS}(w)|$ by Proposition 3.7.

For the converse let $w \in \Sigma^*$ satisfying $|\operatorname{SAS}(w)| = |\operatorname{MAS}(w)|$. Firstly assume there is $a \in \Sigma$ such that $|w|_a > \iota(w)$. Then $a^{|w|_a+1}$ is an MAS of w but not an SAS of w, a contradiction. So every letter occurs exactly $\iota(w)$ times in w and therefore exactly once in each arch $\operatorname{ar}_w(\ell)$. Hence there is a permutation π such that $\pi(\operatorname{ar}_w(1)) = v_1$. Now assume there is an integer $\ell \leq \iota(w)$ such that $\pi(\operatorname{ar}_w(1)\operatorname{ar}_w(2)\cdots\operatorname{ar}_w(\ell)) \neq A_\ell$. Let k be minimal with this property. By definition of π we have $k > 1$ and w.l.o.g we can assume k to be even. Then $\pi(\operatorname{ar}_w(k)) \neq v_2 = \sigma\cdots 21$, that is we can find two positions $i < j \in \operatorname{ar}_w(k)$ such that $\pi(w[i]) < \pi(w[j])$. We claim that $u = w[i]^k w[j]^{\iota(w)-k+2}$ is an MAS but not an SAS of w. We start by checking if $u_1 = w[i]^{k-1}w[j]^{\iota(w)-k+2}$ and $u_2 = w[i]^k w[j]^{\iota(w)-k+1}$ are subsequences of w. We have $\pi(\operatorname{ar}_w(k-1)) = v_1 = 12\cdots\sigma$ and $\pi(w[i]) < \pi(w[j])$ hence $w[i]w[j]$ occurs in $\operatorname{ar}_w(k-1)$. Then $w[i]^{k-1}w[j]$ is a subsequence of $\operatorname{ar}_w(1)\cdots\operatorname{ar}_w(k-1)$ and $w[j]^{\iota(w)-k+1}$ is a subsequence of $\operatorname{ar}_w(k)\cdots\operatorname{ar}_w(\iota(w))$. Similarly $w[i]^k w[j]$ is a subsequence of $\operatorname{ar}_w(1)\cdots\operatorname{ar}_w(k)$ and $w[j]^{\iota(w)-k}$ is a subsequence of $\operatorname{ar}_w(k+1)\cdots\operatorname{ar}_w(\iota(w))$. So u_1, u_2 are subsequences of w and u is an MAS of w. We have $|u| = \iota(w) + 2$ and so u is not an SAS of w, a contradiction. Hence $\pi(\operatorname{ar}_w(1)\cdots\operatorname{ar}_w(\ell)) = A_\ell$ holds for all $\ell \leq \iota(w)$ and therefore $\pi(w) = A_{\iota(w)}$. ☐

In particular, one can show that the number of SAS in the words A_k is exponential in the length of the word. The main idea is to observe that an SAS in A_k is a sequence $i_1, \ldots, i_{k+1}$ of numbers from $\{1, \ldots, \sigma\}$, such that $i_{2\ell-1} \geq i_{2\ell}$ for all $\ell \in [1 : \lceil k/2 \rceil]$ and $i_{2\ell} \leq i_{2\ell+1}$ for all $\ell \in [1 : \lfloor k/2 \rfloor]$. We then can estimate the number of such sequences of numbers and obtain the following result.

Proposition 3.10. The word A_k has an exponential (in $|A_k|$) number of SAS.

Proof:
Again, recall our assumption that σ is a large even constant. An SAS in A_k is a sequence $i_1, \ldots, i_{k+1}$ of numbers from $\{1, \ldots, \sigma\}$, such that $i_{2\ell-1} \geq i_{2\ell}$ for all $\ell \in [1 : \lceil k/2 \rceil]$ and $i_{2\ell} \leq i_{2\ell+1}$ for all $\ell \in [1 : \lfloor k/2 \rfloor]$. Let $S_{t,i}$ be the number of strings of length $t \leq k$ ending with letter $i \in \Sigma$, which can be a prefix of such a sequence (i.e., of an SAS of A_k).

The first observations are the following:

- For even $t \geq 2$ we have that $S_{t,i} = \sum_{j\in[i:\sigma]} S_{t-1,j}$, and, therefore, $S_{t,1} = \sum_{i\in[1:\sigma]} S_{t-1,i}$. The sequence $S_{t,\sigma}, S_{t,\sigma-1}, S_{t,\sigma-2}, \ldots, S_{t,1}$ is increasing.
- For odd $t \geq 3$ we have that $S_{t,i} = \sum_{j\in[1:i]} S_{t-1,j}$, so $S_{t,\sigma} = \sum_{i\in[1:\sigma]} S_{t-1,i}$. The sequence $S_{t,1}, S_{t,2}, \ldots, S_{t,\sigma-1}, S_{t,\sigma}$ is increasing.

Clearly, $S_{1,i} = 1$ for all $i \in \Sigma$.

We claim that for all t we have that $\sum_{i\in[1:\sigma]} S_{t,i} \geq \left(\frac{\sigma}{2}\right)^t$. We will prove this by induction.

This clearly holds for $t = 1$. Assume that it holds for $t \leq q$ and we show that it holds for $t = q+1$.

First, assume that $q+1$ is odd. By the induction hypothesis, we have that $S_{q+1,\sigma} = \sum_{i\in[1:\sigma]} S_{q,i} \geq \left(\frac{\sigma}{2}\right)^q$. Moreover, $S_{q+1,1} = \sum_{i\in[1:1]} S_{q,i} = S_{q,1} = \sum_{i\in[1:\sigma]} S_{q-1,i} \geq \left(\frac{\sigma}{2}\right)^{q-1}$.

Now, we have that:

$$\sum_{i\in[1:\sigma]} S_{q+1,i} = \sum_{j\in[1:\sigma/2]} (S_{q+1,j} + S_{q+1,\sigma+1-j}) = \sum_{j\in[1:\sigma/2]} \left(\sum_{g\in[1:j]} S_{q,g} + \sum_{g\in[1:\sigma-j+1]} S_{q,g} \right),$$

and, therefore,

$$\sum_{i\in[1:\sigma]} S_{q+1,i} = \sum_{j\in[1:\sigma/2]} \left(S_{q,1} + \sum_{g\in[2:j]} S_{q,g} + \sum_{g\in[1:\sigma-j+1]} S_{q,g} \right).$$

Now, as $S_{q,\sigma}, S_{q,\sigma-1}, S_{q,\sigma-2}, \ldots, S_{q,1}$ is increasing, we get that:

$$\sum_{i\in[1:\sigma]} S_{q+1,i} \geq \sum_{j\in[1:\sigma/2]} \left(S_{q,1} + \sum_{g\in[\sigma-j+2:\sigma]} S_{q,g} + \sum_{g\in[1:\sigma-j+1]} S_{q,g} \right).$$

Therefore:

$$\sum_{i\in[1:\sigma]} S_{q+1,i} \geq \sum_{j\in[1:\sigma/2]} \left(S_{q,1} + \sum_{g\in[1:\sigma]} S_{q,g} \right) = \frac{\sigma}{2} \left(S_{q,1} + \sum_{g\in[1:\sigma]} S_{q,g} \right).$$

We can now apply the induction hypothesis, and obtain that

$$\sum_{i\in[1:\sigma]} S_{q+1,i} \geq \frac{\sigma}{2} \left(\left(\frac{\sigma}{2}\right)^{q-1} + \left(\frac{\sigma}{2}\right)^q \right) > \left(\frac{\sigma}{2}\right)^{q+1}.$$

An analogous argument works for the case when $q + 1$ is even.
This concludes our induction proof.

Therefore, the number of strings of length k which can be a prefix of a SAS of A_k is at least $\left(\frac{\sigma}{2}\right)^k$. We note that this is not a tight bound, and a more careful analysis should definitely improve it.

So $|\operatorname{SAS}(A_k)| \geq \left(\frac{\sigma}{2}\right)^k$, which, given that σ is a constant, proves our statement: $|\operatorname{SAS}(A_k)| \geq \left(\frac{\sigma}{2}\right)^{|A_k|/\sigma}$. □

The following few results formalise an additional insightful observation about the word A_k and its set of SAS. We will need the following initial remarks and lemmas.

Lemma 3.11. Let $w, w_1, w_2 \in \Sigma^*$ such that $w = w_1 w_2$, $\iota(w_2) \geq 1$, and $r(w_1) = \varepsilon$.

Then $v[1] \cdots v[\iota(w) + 1]$ is an SAS of w if and only if $v[1] \cdots v[\iota(w_1) + 1]$ is an SAS of w_1 and $v[\iota(w_1) + 1] \cdots v[\iota(w) + 1]$ is an SAS of w_2.

Proof:
Note first that the arches of w coincide with the arches of w_1 and w_2. If $v \in \mathrm{SAS}(w)$ and we greedily choose positions for $v[1] \cdots v[\iota(w_1)]$ in w_1, we find the same positions as if we greedily choose them in w. As $r(w_1) = \varepsilon$, $v[1] \cdots v[\iota(w_1) + 1]$ is an SAS of w_1. Furthermore, let i be the position of $v[\iota(w_1) + 1]$ in $\mathrm{ar}_{w_2}(1)$. Now $v[\iota(w_1) + 2]$ occurs in $\mathrm{ar}_{w_2}(1)[1 : i]$ because it occurs in $\mathrm{ar}_{w_2}(1)$ and if it occurs to the right of i then, by Theorem 3.4, v is not an SAS of w. Hence we can find positions for $v[\iota(w_1) + 1]$ and $v[\iota(w_1) + 2]$ in w_2 satisfying Theorem 3.1. For the remaining letters of v we can find positions in w_2 satisfying Theorem 3.1 because w_2 is a suffix of w and we can find these positions in w. Hence $v[\iota(w_1) + 1 : \iota(w) + 1]$ is an SAS of w_2.

The other implication is trivial. □

Generally, we apply Lemma 3.11 for $w_1 = \mathrm{ar}_w(1) \cdots \mathrm{ar}_w(\ell)$, $w_2 = \mathrm{ar}_w(\ell+1) \cdots \mathrm{ar}_w(\iota(w)) r(w)$, for some $\ell \leq \iota(w)$.

We continue with recalling the so-called rearrangement inequality.

Lemma 3.12. If $a_1, \dots, a_n$ and $b_1, \dots, b_n$ are both increasing (respectively, decreasing) sequences of natural numbers and θ is a permutation of $\{1, 2, \dots n\}$, then $\sum_{i=1}^{n} a_i b_{n+1-i} \leq \sum_{i=1}^{n} a_i b_{\theta(i)} \leq \sum_{i=1}^{n} a_i b_i$.

The following lemma is immediate.

Lemma 3.13. If w' is a subsequence of w and $\iota(w) = \iota(w')$, then $\mathrm{SAS}(w) \subset \mathrm{SAS}(w')$.

Before we begin to show the main observation on the word A_k please recall $v_1 = 12 \cdots \sigma$, $v_2 = \sigma(\sigma - 1) \cdots 1$ and $A_k = w_1 w_2 \cdots w_{k-2} w_{k-1} w_k$, where $w_\ell = v_1$ if ℓ is odd and $w_\ell = v_2$ otherwise. We note $\mathrm{SAS}(v_1) = \{ij \in \Sigma^2 \mid i \geq j\}$ and $\mathrm{SAS}(v_2) = \{ij \in \Sigma^2 \mid i \leq j\}$. That is we have i SAS of v_1 starting with i and $\sigma + 1 - i$ SAS of v_1 ending on i. For v_2 these values are switched: we have $\sigma + 1 - i$ SAS starting with i and i SAS of v_2 ending on i.

Proposition 3.14. $|\mathrm{SAS}(A_k)| \geq |\mathrm{SAS}(w)|$ holds for all $w \in \Sigma^*$ with $\iota(w) = k$.

Proof:
By Lemma 3.13 it suffices to show that Proposition 3.14 holds for words w, such that every arch of w contains every letter exactly once and $r(w) = \varepsilon$. So, in this proof, all words we use have each letter of the alphabet exactly once in each arch. This means for any 1-universal word v there is a permutation π such that $\pi(v) = v_2$.

We will show Proposition 3.14 in the following way: if $\iota(w) = 1$ then there is a permutation π of Σ such that $\pi(w) = v_1 = A_1$, hence $|\mathrm{SAS}(w)| = |\mathrm{SAS}(A_1)|$. For general $\iota(w) \geq 2$ we let $w_\ell = \mathrm{ar}_w(1)\,\mathrm{ar}_w(2) \cdots \mathrm{ar}_w(\iota(w) - \ell)$ (with $\iota(w_\ell) = \iota(w) - \ell$) and show that there exists a permutation π of Σ such that $|\mathrm{SAS}(w)| \leq |\mathrm{SAS}(\pi(w_\ell) A_\ell)|$ for every word w and integer $2 \leq \ell \leq \iota(w)$. Inductively this shows Proposition 3.14.

Firstly let $f_i(w) = |\Sigma^* i \cap \mathrm{SAS}(w)|, g_i(w) = |i\Sigma^* \cap \mathrm{SAS}(w)|$ and $h_{i,j}(w) = |i\Sigma^* j \cap \mathrm{SAS}(w)|$. That is, $f_i, g_i, h_{i,j}$ denote the respective number of SAS of w which end on i, start with i, and start with i and end with j respectively. Then, for any factorisation $w = u_1u_2$ satisfying Lemma 3.11 we have $|\mathrm{SAS}(w)| = |\mathrm{SAS}(u_1u_2)| = \sum_{i\in\Sigma} f_i(u_1)g_i(u_2)$. For $w \in \Sigma^*$ and v_1, v_2, A_k as above, we have $f_i(wv_1) = \sum_{j\geq i} f_j(w)$ and $g_i(v_2A_k) = \sum_{j\geq i} g_j(A_k)$ because $h_{j,i}(v_1) = h_{i,j}(v_2) = 1$ if $j \geq i$ and 0 otherwise. Furthermore $f_i(w) = f_{\pi(i)}(\pi(w))$, $g_i(w) = g_{\pi(i)}(\pi(w))$ and $h_{i,j}(w) = h_{\pi(i),\pi(j)}(\pi(w))$ by Lemma 3.8; when using this observation, we will refer to it using (*).

We start by showing that the proposition holds for $\ell = 2$. Let $\iota(w) = k \geq 2$, π be a permutation of Σ such that $\pi(\mathrm{ar}_w(k-1)) = v_1$ and $v' = \pi(\mathrm{ar}_w(k))$ then

$$
\begin{aligned}
|\mathrm{SAS}(w)| &= |\mathrm{SAS}(\pi(w))| = |\mathrm{SAS}(\pi(w_2)v_1v')| \\
&= \sum_{i\in\Sigma} f_i(\pi(w_2)v_1)g_i(v') = \sum_{i\in\Sigma} f_i(\pi(w_2)v_1)g_{\tau(i)}(v_2) \\
&\leq \sum_{i\in\Sigma} f_i(\pi(w_2)v_1)g_i(v_2) = |\mathrm{SAS}(\pi(w_2)v_1v_2)| && \text{(by Lemma 3.12)} \\
&= |\mathrm{SAS}(\pi(w_2)A_2)|,
\end{aligned}
$$

where τ denotes the permutation which maps v' to v_2. Please note that $g_i(v_2) = |\{ij \mid i,j \in \Sigma, j \geq i\}|$ and $f_i(\pi(w_2)v_1) = \sum_{j\geq i} f_j(\pi(w_2))$ are decreasing (as sequences w.r.t. i).

Now, we assume we have that for every $w \in \Sigma^*$ and $\ell < \iota(w)$ we can find a permutation π of Σ such that $|\mathrm{SAS}(w)| \leq |\mathrm{SAS}(\pi(w_\ell)A_\ell)|$. We show that the result holds for $\ell + 1$ as well; that is, we show that $|\mathrm{SAS}(w)| \leq |\mathrm{SAS}(\pi''(w_{\ell+1})A_{\ell+1})|$ for some permutation π''. Indeed, we have:

$$
\begin{aligned}
|\mathrm{SAS}(w)| &\leq |\mathrm{SAS}(\pi(w_\ell)A_\ell)| = |\mathrm{SAS}(\pi(w_{\ell+1})\pi(\mathrm{ar}_w(\iota(w)-\ell))A_\ell)| \\
&= \sum_{i\in\Sigma} f_i(\pi(w_{\ell+1})) \left(\sum_{j\in\Sigma} h_{i,j}(\pi(\mathrm{ar}_w(\iota(w)-\ell)))g_j(A_\ell)\right) \\
&= \sum_{i\in\Sigma} f_i(\pi(w_{\ell+1})) \left(\sum_{j\in\Sigma} h_{\tau(i),\tau(j)}(v_2)g_j(A_\ell)\right) && \text{(by (*))} \\
&\leq \sum_{i\in\Sigma} f_i(\pi(w_{\ell+1})) \left(\sum_{j\in\Sigma} h_{\tau(i),j}(v_2)g_j(A_\ell)\right) && \text{(by Lemma 3.12)} \\
&= \sum_{i\in\Sigma} f_{\tau(i)}(\tau(\pi(w_{\ell+1}))) \left(\sum_{j\in\Sigma} h_{\tau(i),j}(v_2)g_j(A_\ell)\right) && \text{(by (*))} \\
&= \sum_{\tau(i)\in\Sigma} f_i(\tau(\pi(w_{\ell+1}))) \left(\sum_{j\in\Sigma} h_{i,j}(v_2)g_j(A_\ell)\right) \\
&= \sum_{i\in\Sigma} f_i(\tau(\pi(w_{\ell+1}))) \left(\sum_{j\in\Sigma} h_{i,j}(v_2)g_j(A_\ell)\right) \\
&= |\mathrm{SAS}(\tau(\pi(w_{\ell+1}))\pi'(A_{\ell+1}))| = |\mathrm{SAS}(\pi''(w_{\ell+1})A_{\ell+1})|,
\end{aligned}
$$

where τ is a permutation mapping $\mathrm{ar}_w(\iota(w)-\ell)$ to v_2, π' is the permutation which maps i to $\sigma+1-i$ (and therefore also maps v_1 to v_2) and $\pi'' = (\pi')^{-1} \circ \tau \circ \pi$. The second inequality in the reasoning above follows from Lemma 3.12 because $g_j(A_k)$ and $h_{\tau(i),j}(v_2)$ both are monotonically increasing as sequences in j. This concludes our proof. □

In the conclusion of this section, we make the following remarks. Propositions 3.6 and 3.10 motivate our investigation for compact representations of the sets of SAS and MAS of words. These sets can be exponentially large, and we would still like to have efficient (i.e., polynomial) ways of representing them, allowing us to explore and efficiently search these sets.

4. Algorithms

The results we present from now on are of algorithmic nature. The computational model we use to describe our results is the standard unit-cost RAM with logarithmic word size: for an input of size n, each memory word can hold $\log n$ bits. Arithmetic and bitwise operations with numbers in $[1:n]$ are, thus, assumed to take $O(1)$ time. Numbers larger than n, with ℓ bits, are represented in $O(\ell/\log n)$ memory words, and working with them takes time proportional to the number of memory words on which they are represented. 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 $|\Sigma| = \sigma \leq n$. That is, we assume that the processed words are sequences of integers (called letters or symbols, each fitting in $O(1)$ memory words). On the one hand, note that we no longer assume that σ is a constant, as in the combinatorial results of the previous section. On the other hand, assuming that $\sigma \in O(n)$ is a common assumption in string algorithms: the input alphabet is said to be *an integer alphabet*, see also the discussion in, e.g., [55]. For its use in the particular case of algorithms related to subsequences see, for instance, [33, 34].

In all the problems, we assume that we are given a word w of length n over an *integer alphabet* $\Sigma = \{1, 2, \ldots, \sigma\}$, with $|\Sigma| = \sigma \leq n$. As the problems considered here are trivial for unary alphabets, we also assume $\sigma \geq 2$.

We start with some preliminaries and simple initial results. The decomposition of word w into its arches can be done with a greedy approach. The following theorem is well known and a proof can be seen for example in [32]. It shows that the universality index and the decomposition into arches can be obtained in linear time.

Theorem 4.1. Given a word w, of length n, we can compute the universality index $\iota(w)$, the arch factorisation $\mathrm{ar}_w(1) \cdots \mathrm{ar}_w(\iota(w))\, \mathrm{r}(w)$ of w, as well as the set $\Sigma \setminus \mathrm{alph}(r(w))$ of letters which do not occur in $\mathrm{r}(w)$ in linear time $O(n)$.

Proof:
One can compute greedily the decomposition of w into arches $w = \mathrm{ar}_w(1) \cdots \mathrm{ar}_w(k)\, \mathrm{r}(w)$ in linear time as follows.

- $\mathrm{ar}_w(1)$ is the shortest prefix of w with $\mathrm{alph}(\mathrm{ar}_w(1)) = \Sigma$, or $\mathrm{ar}_w(1) = w$ if there is no such prefix;

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

The concrete computation can be also seen in Algorithm 1. Note that the complexity is linear because the re-initialization of C is amortized by the steps in which the elements of C were incremented.

Algorithm 1: computeArchFactorisation(w, Σ)

Input: Word w with $|w| = n$, alphabet $\Sigma = \mathrm{alph}(w)$ with $|\Sigma| = \sigma$
Output: Variables $n_1, \ldots, n_k$ containing the last position of the respective arch

```
1  define array C of length σ with elements initially set to 0;
   /* Array C is used for marking letters as already encountered        */
2  define counter h ← σ;
   /* Counter h is used to check if all letters of the alphabet have been
      encountered                                                      */
3  define arch counter k ← 1;
4  for i = 1 to n do
5  |  if C[w[i]] == 0 then
6  |  |  h ← h − 1;
7  |  |  C[w[i]] ← 1;
8  |  if h == 0 then
9  |  |  n_k ← i;
10 |  |  k ← k + 1;
11 |  |  h ← σ;
12 |  |  reset array C by initialising all elements with 0 again;
13 return n_1 to n_k;
```

Further, we can simply define a binary array f indexed by Σ, whose components are initially set to 0. We then go through $\mathrm{r}(w)$, and set $f[a] = 1$ if and only if a appears in $\mathrm{r}(w)$. Clearly, the set of letters which does not occur in $\mathrm{r}(w)$ (that is, $\Sigma \setminus \mathrm{alph}(r(w))$) is the set of letters a with $f[a] = 0$. A list with these letters can be computed in $O(\sigma)$ time. We now only have to note that this processing takes at most $O(n)$ time. This concludes our proof. □

Remark 4.2. For a word w with length n, further helpful notations are $\mathrm{next}_w(a,i)$, which denotes the next occurrence of letter a in the word $w[i:n]$, and $\mathrm{last}_w(a,i)$, which denotes the last occurrence of letter a in the word $w[1:i]$. Both these values can be computed by simply traversing w from position i to the right or left, respectively. The runtime of computing $\mathrm{next}_w(a,i)$ is proportional to the length of the shortest factor $w[i:j]$ of w such that $w[j] = a$ as we only traverse the positions from word w once from i towards the right until we meet the first letter a. The runtime of $\mathrm{last}_w(a,i)$ is proportional to the length of the shortest factor $w[j:i]$ of w such that $w[j] = a$ as we only traverse the positions from word w once from i towards the left until we meet the first letter a.

Proof:
See Line 2 and Algorithm 3.

Algorithm 2: $\text{next}_w(a, i)$

```
  Input: Word w with |w| = n, letter a, position i
  Result: Position of next occurence from letter a starting from position i in word w
1 if i < 1 or i > n then
2 |  return ∞;
3 for j = i to n do
4 |  if w[j] == a then
5 |  |  return j;
6 return ∞;
```

Algorithm 3: $\text{last}_w(a, i)$

```
  Input: Word w with |w| = n, letter a, position i
  Result: Position of last occurence from letter a starting from position i in word w
1 if i < 1 or i > n then
2 |  return ∞;
3 for j = i to 1 do
4 |  if w[j] == a then
5 |  |  return j;
6 return ∞;
```

□

Based on these notations, we can define the function $\text{isSubseq}(w, u)$ which checks if a word u is a subsequence of a word w. This can be done by utilizing a greedy approach as discussed in Section 3 and depicted in Algorithm 4. While this approach is standard and relatively straightforward, it is important to note it before proceeding with our algorithms. The idea is the following. We consider the letters of u one by one, and try to identify the shortest prefix $w[1 : i_j]$ of w which contains $u[1 : j]$ as a subsequence. To compute the shortest prefix $w[1 : i_{j+1}]$ of w which contains $u[1 : j + 1]$, we simply search for the first occurrence of $u[j+1]$ in w after position i_j. The runtime of the algorithm isSubseq

Algorithm 4: isSubseq(w,u)

```
  Input: Word w with |w| = n, word u with |u| = m to be tested
1 pos ← 0;
2 for i = 1 to m do
3 |  pos ← next_w(u[i], pos + 1);
4 return pos == ∞ ? false : true;
```

is clearly linear in the worst case. When u is a subsequence of w, then $w[1 : pos]$ is the shortest prefix of w which contains u, and the runtime of the algorithm is $O(pos)$.

A further helpful notation is $\mathrm{llo}(w) = \min\{\mathrm{last}_w(a, n) \mid a \in \Sigma\}$, the position of the leftmost of the last occurrences of the letters of Σ in w. The following lemma is not hard to show.

Lemma 4.3. Given a word w of length n, we can compute $\mathrm{llo}(w)$ in $O(n)$ time.

Proof:
See Algorithm 5. This is clearly correct and runs in linear time.

Algorithm 5: $\mathrm{llo}(w, \Sigma)$

```
  Input: Word w with |w| = n, alphabet Σ = alph(w) with |Σ| = σ
  Output: Leftmost position in word w of all last occurences of the letters in Σ
1 define array C of length σ with elements initially set to 0;
  /* Array C is used for marking letters as already encountered        */
2 define counter h ← σ;
3 for i = n to 1 do
4 |  if C[w[i]] == 0 then
5 |  |  C[w[i]] ← 1;
6 |  |  h ← h − 1;
7 |  |  if h == 0 then
8 |  |  |  return i;
9 return ∞;
```

□

Based on these notions, we can already present our first results, regarding the basic algorithmic properties of SAS and MAS. Firstly, we can easily compute an SAS (and, therefore, an MAS) in a given word.

Lemma 4.4. Given a word w of length n with $\iota(w) = k$, its arch decomposition (i.e., the last position of each arch), and the set $\Sigma \setminus \mathrm{alph}(r(w))$ (i.e., as a list of letters), we can retrieve in $O(k)$ time an SAS (and, therefore, an MAS) of w.

Proof:
By Definition 2.3, the word $m(w) = \mathrm{ar}_w(1)[|\mathrm{ar}_w(1)|] \cdots \mathrm{ar}_w(k)[|\mathrm{ar}_w(k)|]$ contains the unique last letters of each arch. By appending a letter not contained in $\mathrm{r}(w)$ to $m(w)$, we achieve a word that is an SAS satisfying the properties from Theorem 3.4. Algorithm 6 shows how to construct $m(w)$ in $O(k)$ time. To extend $m(w)$ to an SAS, we have to append to the existing word a letter that is not contained in $\mathrm{r}(w)$, which can be done $O(1)$ if that set is given as a list. □

The following theorem shows that we can efficiently test if a given word u is an SAS or MAS of a given word w.

Algorithm 6: getSAS(w)

Input: Word with its arch factorization $w = \mathrm{ar}_w(1)\cdots\mathrm{ar}_w(k)\,\mathrm{r}(w)$ with $|w| = n$
1 define empty word sas;
2 **for** $i = 1$ **to** k **do**
3   add the last letter of arch $\mathrm{ar}_w(i)$ to the end of sas:
4   sas = sas $\cdot\, \mathrm{ar}_w(i)[|\,\mathrm{ar}_w(i)|]$;
5 add a letter of the alphabet that does not occur in $\mathrm{r}(w)$ to the end of sas
6 **return** sas;

Theorem 4.5. Given a word w of length n and a word u of length m, we can test in $O(n)$ time whether u is an SAS or MAS of w.

Proof:
We recall that by definition a word u is an SAS of a word w if and only if $|v| = \iota(w)+1$ and v is absent. It is therefore sufficient to check the length of an SAS candidate and whether it is a subsequence of w or not. In Algorithm 7 we can now see this implemented with a runtime in $O(n)$.

Algorithm 7: isSAS(w,u)

Input: Word with its arch factorization $w = \mathrm{ar}_w(1)\cdots\mathrm{ar}_w(k)\,\mathrm{r}(w)$, word u (over the same alphabet as w) to be tested
1 **if** $|u|$ = $k+1$ **and** `!isSubseq(`w, u`)` **then**
2   **return** true;
3 **else**
4   **return** false;

To decide if u is an MAS, we can check whether u is indeed an absent subsequence and every subsequence of u of length $m-1$ is also a subsequence of w. In other words, u is turned into a subsequence of w by removing any of its letters. It is important to note at this point that the subsequences of length $m-1$ of u have the form $u[1:i]u[i+2:m]$ for some $i \leq m-1$.

Based on this observation, we first compute using Algorithm 8 the shortest prefixes of w containing $u[1:1], u[1:2], \ldots, u[1:m]$ and the shortest suffixes of w containing $u[m:m], u[m-1:m], \ldots, u[1:m]$. With the knowledge over these prefixes and suffixes, we can quickly decide for a position i of u, if $u[1:i-1]u[i+1:m]$ is a subsequence or an absent subsequence of w. For this, we check if the shortest prefix of w containing $u[1:i-1]$ is overlapping with the shortest suffix containing $u[i+1:m]$. If they are overlapping, we obtained from u an absent subsequence $u[1:i-1]u[i+1:m]$ by removing the letter $u[i]$, meaning that u is not an MAS.

If, for all positions $i \in [1:m]$, the shortest prefix containing $u[1:i-1]$ and the shortest suffix containing $u[i+1:m]$ are not overlapping, then u is clearly an MAS: all its subsequences of length $m-1$ are also subsequences of w, while u is absent. Checking whether this property holds takes linear time. Algorithm 8 is an implementation in pseudocode of the strategy presented in the proof. □

Algorithm 8: isMAS(w,u)

Input: Word with its arch factorization $w = \mathrm{ar}_w(1) \cdots \mathrm{ar}_w(k)\, \mathrm{r}(w)$ with $|w| = n$, word u (over the same alphabet as w) to be tested with $|u| = m$

```
1  if |u| < k + 1 then return false;
2  if isSubseq(w,u) then return false;
3  isMAS ← true;
4  define array L of size m;
5  define array R of size m;
   /* we want to fill the arrays L and R so that L[i] holds the end
      position of the shortest prefix of w containing the subsequence
      u[1 : i], and R[i] holds the start position of the shortest suffix of
      w containing the subsequence u[i : m]                              */
6  L[1] ← next_w(u[1], 1);
7  for i = 2 to m do
8  |   L[i] ← next_w(u[i], L[i − 1] + 1);
9  R[m] ← last_w(u[m], n);
10 for i = m − 1 to 1 do
11 |   R[i] ← last_w(u[i], R[i + 1] − 1);
12 if !isSubseq(u[1 : m − 1]) or !isSubseq(u[2 : m]) then
13 |   return false;
14 for i = 1 to m − 2 do
15 |   if L[i] >=R[i + 2] then
16 |   |   return false;
17 return isMAS;
```

4.1. A compact representation of the SAS of a word

We now introduce a series of data structures which are fundamental for the efficient implementation of the main algorithms presented in this paper.

For a word w of length n with arch factorization $w = \mathrm{ar}_w(1) \cdots \mathrm{ar}_w(k)\, \mathrm{r}(w)$, we define two $k \times \sigma$ arrays $\mathrm{firstInArch}[\cdot][\cdot]$ and $\mathrm{lastInArch}[\cdot][\cdot]$ by the following relations. For $\ell \in [1 : k]$ and $a \in \Sigma$, $\mathrm{firstInArch}[\ell][a]$ is the leftmost position of $\mathrm{ar}_w(\ell)$ where a occurs and $\mathrm{lastInArch}[\ell][a]$ is the rightmost position of $\mathrm{ar}_w(\ell)$ where a occurs. These two arrays are very intuitive: they simply store for each letter of the alphabet its first and last occurrence inside the arch. To avoid confusions, for all ℓ and a, the values stored in $\mathrm{firstInArch}[\ell][a]$ and $\mathrm{lastInArch}[\ell][a]$ are positions of w (so, between 1 and $|w|$), they are not defined as positions of $\mathrm{ar}_w(\ell)$.

Example 4.6. For $w = 12213.113312.21$, with arches $\mathrm{ar}_w(1) = 12213$ and $\mathrm{ar}_w(2) = 113312$ and rest $\mathrm{r}(w) = 21$, we have the following.

	$\mathrm{firstInArch}[\cdot][1]$	$\mathrm{firstInArch}[\cdot][2]$	$\mathrm{firstInArch}[\cdot][3]$
$\mathrm{firstInArch}[1][\cdot]$	1	2	5
$\mathrm{firstInArch}[2][\cdot]$	6	11	8

	$\mathrm{lastInArch}[\cdot][1]$	$\mathrm{lastInArch}[\cdot][2]$	$\mathrm{lastInArch}[\cdot][3]$
$\mathrm{lastInArch}[1][\cdot]$	4	3	5
$\mathrm{lastInArch}[2][\cdot]$	10	11	9

Lemma 4.7. For a word w of length n, we can compute $\mathrm{firstInArch}[\cdot][\cdot]$ and $\mathrm{lastInArch}[\cdot][\cdot]$ in $O(n)$ time.

Proof:
As illustrated by Algorithm 9, we can traverse the word w from left to right, and similar to the computation of the arch factorisation, we keep track of the first encounters of each letter in each arch. The entries for the $\mathrm{lastInArch}[\cdot]$ array will then be updated continuously until the end of an arch is reached.

Algorithm 9: $\mathrm{firstInArch}_w(a, i)$ and $\mathrm{lastInArch}_w(a, i)$

Input: Word with its arch factorization $w = \mathrm{ar}_w(1) \cdots \mathrm{ar}_w(k)\,\mathrm{r}(w)$ with $|w| = n$, alphabet Σ with $|\Sigma| = \sigma$
Result: Arrays $\mathrm{firstInArch}$ und $\mathrm{lastInArch}$ for word w

1 define array $\mathrm{firstInArch}[1 : k][1 : \sigma]$;
2 define array $\mathrm{lastInArch}[1 : k][1 : \sigma]$;
3 define array C of size σ;
4 current position $posW \leftarrow 1$;
5 **for** $i = 1$ **to** k **do**
6   reset all elements of C to 0;
7   **foreach** *letter* a *in* $\mathrm{ar}_w(i)$ **do**
8   **if** $C[a] == 0$ **then**
9             $C[a] \leftarrow 1$;
10            $\mathrm{firstInArch}[i][a] = posW$;
11            $\mathrm{lastInArch}[i][a] = posW$;
12   **else**
13            $\mathrm{lastInArch}[i][a] = posW$;
14        $posW \leftarrow posW + 1$;
15 **return** $\mathrm{firstInArch}$ **and** $\mathrm{lastInArch}$;

Note that Algorithm 9 can be easily combined with Algorithm 1 to compute the arch factorisation and the arrays $\mathrm{firstInArch}[\cdot]$ and $\mathrm{lastInArch}[\cdot]$ while traversing the word w once from right to left.

$\square$

We continue with the array $\mathrm{minArch}[\cdot]$ with n-elements, where, for $i \in [1:n]$, $\mathrm{minArch}[i] = j$ if and only if $j = \min\{g \mid \mathrm{alph}(w[i:g]) = \Sigma\}$. If $\{g \mid \mathrm{alph}(w[i:g]) = \Sigma\} = \emptyset$, then $\mathrm{minArch}[i] = \infty$. Intuitively, $\mathrm{minArch}[i]$ is the end point of the shortest 1-universal factor (i.e., arch) of w starting at i.

Example 4.8. Consider the word $w = 12213.113312.21$ with the arches $\mathrm{ar}_w(1) = 12213$ and $\mathrm{ar}_w(2) = 113312$ and the rest $\mathrm{r}(w) = 21$. We have $\mathrm{minArch}[j] = 5$ for $j \in [1:3]$, $\mathrm{minArch}[j] = 11$ for $j \in [4:9]$, and $\mathrm{minArch}[j] = \infty$ for $j \in [10:13]$.

The array $\mathrm{minArch}[\cdot]$ can be computed efficiently.

Lemma 4.9. For a word w of length n, we can compute $\mathrm{minArch}[\cdot]$ in $O(n)$ time.

Proof:
The algorithm starts with a preprocessing phase in which we define an array count$[\cdot]$ of size $|\Sigma|$ and initialize its components with 0. We also initialize all elements of the the array $\mathrm{minArch}[\cdot]$ with ∞.

In the main phase of our algorithm, we use a 2 pointers strategy. We start with both p_1 and p_2 on the first position of w. Then, the algorithm consists in an outer loop which has two inner loops. The outer loop is executed while $p_2 \leq n$. We describe in the following one iteration of the outer loop. We start by executing the first inner loop where, in each iteration, we increment repeatedly p_2 by 1 (i.e., advance it one position to the right in the word w), until the factor $w[p_1 : p_2]$ contains all letters of the alphabet (i.e., it is 1-universal). If p_2 becomes $n+1$, we simply stop the execution of this loop. Then, the second inner loop is executed only if $p_2 \leq n$. In each iteration of this loop, while the factor $w[p_1 : p_2]$ contains all the letters of the alphabet, we set $\mathrm{minArch}[p_1] \leftarrow p_2$, and we advance p_1 one position towards the right. This second loop stops as soon as $w[p_1 : p_2]$ does not contain all the letters of the alphabet anymore. Then, we continue to the next iteration of the outer loop with the current values of p_1 and p_2 as computed in this iteration, but only if $p_2 \leq n$.

The pseudo-code of this algorithm is described in Algorithm 10.

The correctness can be shown as follows.

Firstly, it is clear that at the beginning of the execution of an iteration of the outer loop $w[p_1 : p_2 - 1]$ is not 1-universal. Thus, the same property holds for all of the prefixes and suffixes of $w[p_1 : p_2 - 1]$. Now, let a be the value of p_1 at this point (at the beginning of the execution of an iteration of the outer loop). Then, in the first inner loop we identify the smallest value of $b = p_2$ such that $w[a : b] = w[p_1 : p_2]$ is 1-universal, and it is correct to set $\mathrm{minArch}[a] \leftarrow b$. In the second inner loop we increment p_1 to identify all the suffixes $w[p_1 : b]$ of u which are 1-universal. None of these strings has a proper prefix which is also 1-universal, as this would imply that $\mathrm{minArch}[a] < b$, a contradiction. So, for all the values p_1 visited in this loop it is correct to set $\mathrm{minArch}[p_1] \leftarrow b$. Finally, once we have found the largest p_1 such that $w[p_1 : b]$ is 1-universal, we increment p_1 and p_2 (i.e., p_2 becomes $b+1$) and restart from line 5. This is correct as: $\mathrm{minArch}[j]$ is correctly set for $j < p_1$ and $w[p_1 : p_2 - 1]$ is not 1-universal, so the same reasoning as above applies.

Now assume that there exists a position i with $j = \min\{g \mid \mathrm{alph}(w[i:g]) = \Sigma\}$ and we did not set $\mathrm{minArch}[i] = j$. This means that when p_2 was set to be j during the execution of our algorithm, we had $p_1 > i$. In order for this to hold, our algorithm should have already set $\mathrm{minArch}[i] = j' < j$. As shown above, this means that $w[i : j']$ is 1-universal, a contradiction with the definition of j.

Algorithm 10: minArch(w)

```
   Input: Word w with |w| = n, alphabet Σ with |Σ| = σ
   Output: Array minArch for word w
 1 define array count of length σ with elements initially set to 0;
 2 define array minArch of length n with elements initially set to ∞;
 3 alph ← 0, p1 ← 1, p2 ← 1;
 4 while p2 ≤ n do
 5     while alph ≠ σ and p2 ≠ n + 1 do
 6         count[w[p2]] ← count[w[p2]] + 1;
 7         if count[w[p2]] == 1 then
 8             alph ← alph +1;
 9         if alph < σ then
10             p2 ← p2 + 1;
11     if p2 == n + 1 then
12         return minArch;
13     while alph == σ do
14         minArch[p1] ← p2;
15         count[w[p1]] ← count[w[p1]] − 1;
16         if count[w[p1]] == 0 then
17             alph ← alph −1;
18         p1 ← p1 + 1;
19     p2 ← p2 + 1;
20 return minArch;
```

Further, we note that if the algorithm returns $\mathrm{minArch}$ in line 12, then it can only do so because p_2 was incremented in line 10 (so alph $< \sigma$). This means, that no new arch was found starting on p_1 and ending on a position $\leq n$. Therefore, $\mathrm{minArch}$ is correctly filled in. There is another possibility for the algorithm to end: we set $p_2 = n + 1$ in line 19, and the outer while-loop is not executed anymore. As explained above, we have that $\mathrm{minArch}[j]$ is correctly set for $j < p_1$, and there is no 1-universal word contained in $w[p_1 : n]$ (so $\mathrm{minArch}[j]$ is also correctly set for $j \geq p_1$).

Now, let us discuss the complexity of the algorithm. The algorithm includes a preprocessing phase, which takes $O(n)$ time.

Each of the pointers p_1 and p_2 visits each position of w exactly once, and in each step of the algorithm we increment one of these pointers. This incrementation is based on the result returned by a check whether $w[p_1 : p_2]$ contains all the letters of the alphabet or not. This check can be done in constant time by maintaining an array which counts how many copies of each letter of the alphabet exist in the factor $w[p_1 : p_2]$ and, based on it, a counter keeping track of $|\,\mathrm{alph}(w[p_1 : p_2])|$.

So, overall, the complexity of this algorithm is linear. □

A very important consequence of Lemma 4.9 is that we can define a tree-structure of the arches (i.e., the 1-universal factors occurring in a word). Recall that we have defined $\mathrm{llo}(w) = \min\{\mathrm{last}_w(a, n) \mid a \in \Sigma\}$. That is, $\mathrm{llo}(w)$ is the position of the leftmost of the last occurrences of the letters of Σ in w, and can be computed in linear time by Lemma 4.3. It is immediate to see that the letter $w[\mathrm{llo}(w)]$ does not occur in $\mathrm{r}(w)$.

Definition 4.10. Let w be a word of length n over Σ. The arch-tree of w, denoted by $\mathcal{A}_w$, is a rooted labelled tree defined as follows:

- The set of nodes of the tree is $\{i \mid 0 \leq i \leq n+1\}$, where the node $n+1$ is the root of the tree $\mathcal{A}_w$. The root node $n+1$ is labelled with the letter $w[\mathrm{llo}(w)]$, the node i is labelled with the letter $w[i]$, for all $i \in [1:n]$, and the node 0 is labelled with $\uparrow$.
- For $i \in [1:n-1]$, we have two cases:
  - if $\mathrm{minArch}[i+1] = j$ then node i is a child of node j.
  - if $\mathrm{minArch}[i+1] = \infty$ then node i is a child of the root $n+1$.
- Node n is a child of node $n+1$.

Example 4.11. For the word $w = 12213.113312.21$, the root 14 of $\mathcal{A}$ has the children $9, 10, 11, 12, 13$ and is labelled with 3, the node 11 has the children $3, 4, 5, 6, 7, 8$, and the node 5 has the children $0, 1, 2$. So, the root 14 and the nodes 11 and 5 are internal nodes, while all other nodes are leaves.

Theorem 4.12. For a word w of length n, we can construct $\mathcal{A}_w$ in $O(n)$ time.

Proof:
This is a straightforward consequence of Lemma 4.9. □

Constructed in a straightforward way based on Lemma 4.9, the arch-tree $\mathcal{A}_w$ encodes all the arches occurring in w. Now, we can identify an SAS in $w[i:n]$ by simply listing (in the order from $i-1$ upwards to $n+1$) the labels of the nodes met on the path from $i-1$ to $n+1$, without that of node $i-1$.

For a word w and each $i \in [0:n-1]$, we define $\mathrm{depth}(i)$ as the number of edges on the shortest path from i to the root of A_w. In this case, for $i \in [0:n]$, we have that $w[i+1:n]$ has universality index $\mathrm{depth}(i) - 1$.

Example 4.13. For the word $w = 12213.113312.21$ (same as in Example 4.11), an SAS in $w[5:13]$ contains the letters $w[11] = 2$ and the label 3 of 14, so 23 (corresponding to the path $4 \to 11 \to 14$). An SAS in $w[2:13]$ contains, in order, the letters $w[5] = 3, w[11] = 2$ and the label 3 of 14, so 323 (corresponding to the path $1 \to 5 \to 11 \to 14$). We also have $\mathrm{depth}(4) = 2$, so $\iota(w[5:13]) = 1$, while $\mathrm{depth}(1) = 3$ and $\iota(w[2:13]) = 2$.

Enhancing the construction of $\mathcal{A}_w$ from Theorem 4.12 with level ancestor data structures [56] for this tree, we can now efficiently process *internal* SAS *queries* for a given word w, i.e., we can efficiently retrieve a compact representation of an SAS for each factor of w.

Theorem 4.14. For a word w of length n we can construct in $O(n)$ time data structures allowing us to answer in $O(1)$ time queries $\mathrm{sasRange}(i, j)$*: "return a representation of an* SAS *of* $w[i : j]$*".*

Proof:
We start with the preprocessing phase, in which we construct the data structures allowing us to answer the queries efficiently.

We first construct the tree $\mathcal{A}_w$ for the word w. For each node i of A_w, we compute $\mathrm{depth}(i)$. This can be done in linear time in a standard way.

Then we construct a solution for the *Level Ancestor Problem* for the rooted tree $\mathcal{A}_w$. This is defined as follows (see [56]). For a rooted tree T, $\mathrm{LA}_T(u, d) = v$ where v is an ancestor of u and $\mathrm{depth}(v) = d$, if such a node exists, or $\uparrow$ otherwise. The *Level Ancestor Problem* can now be formulated.

- Preprocessing: A rooted tree T with N vertices. ($T = A_w$ in our case and $N = n + 1$)
- Querying: For a node u in the rooted tree T, query $\mathrm{levelAncestor}_T(u, d)$ returns $\mathrm{LA}_T(u, d)$, if it exists, and false otherwise.

A simple and elegant solution for this problem which has $O(N)$ preprocessing time and $O(1)$ time for query-answering can be found in, e.g., [56] (see also [57] for a more involved discussion). So, for the tree A_w we can compute in $O(n)$ time data structures allowing us to answer $\mathrm{levelAncestor}_{A_w}$ queries in $O(1)$ time.

This is the entire preprocessing we need in order to be able to answer sasRange-queries in constant time.

We will now explain how an sasRange-query is answered.

Let us assume we have to answer $\mathrm{sasRange}(i, j)$, i.e., to return a representation of an SAS of $w[i : j]$. The compact representation of an SAS of $w[i : j]$ will consist in two nodes of the tree A_w and in the following we explain both how to compute these two nodes, and what is their semantics (i.e., how an SAS can be retrieved from them).

Assume that $\mathrm{depth}(i - 1) = x$ and $\mathrm{depth}(j) = y$. We retrieve the ancestor t of the node $i - 1$ which is at distance $x - y$ from this node. So t is the ancestor of depth y of node $i - 1$, and can be retrieved as $\mathrm{levelAncestor}_{A_w}(i - 1, y)$. We check whether $t > j$ (i.e., $w[i : j]$ is a prefix of $w[i : t]$). If $t > j$, then we set t' to be the successor of t on the path to $i - 1$. If $t \leq j$, we set $t' = t$.

The answer to $\mathrm{sasRange}(i, j)$ is the pair of nodes $(i-1, t')$. This answer can be clearly computed in constant time after the preprocessing we have performed.

We claim that this pair of nodes is a compact representation of an SAS of $w[i : j]$, and such an SAS can be obtained as follows: we go in the tree A_w from the node $i-1$ on the path towards node t' and output, in order, the labels of the nodes we meet (except the label of node $i - 1$). Then we output the label of the parent node of t'. This is an SAS of $w[i : j]$.

Let us now explain why the above claim holds.

We recall the following basic fact: if u and v are words, then $\iota(u)+\iota(v) \leq \iota(uv) \leq \iota(u)+\iota(v)+1$. That is, $\iota(u) \in \{\iota(uv) - \iota(v), \iota(uv) - \iota(v) - 1\}$.

Now, from the fact that $\mathrm{depth}(i-1) = x$ and $\mathrm{depth}(j) = y$ we get that $\iota(w[i:n]) = x-1$ and $\iota(w[j+1:n]) = y-1$. Therefore, from the fact recalled above (with $u = w[i:j]$ and $v = w[j+1:n]$), we have that $(x-1)-(y-1)-1 = x-y-1 \le \iota(w[i:j]) \le (x-1)-(y-1) = x-y$. Thus, we compute $w[i:t]$, the shortest factor of w starting on position i which is $x-y$ universal; clearly, $t = \mathrm{levelAncestor}_{A_w}(i-1, y)$. Now, there are two possibilities. In the first case, $\iota(w[i:j]) = x-y-1$ and $j < t$. Then, for t' the successor of t on the path to $i-1$, we have that $w[i:t']$ is the shortest $(x-y-1)$-universal prefix of $w[i:j]$ and $w[t]$ is a letter which does not appear in $w[t'+1:j]$. In the second case, $\iota(w[i:j]) = x-y$ and $j \ge t$. Note that we also have that $\iota(w[i:t]) = x-y$, so j cannot be the parent of t (otherwise, we would have $\iota(w[i:j]) = x-y+1$). Then, in this case, we can take $t' = t$ and we have that $w[i:t'] = w[i:t]$ is the shortest $(x-y)$-universal prefix of $w[i:j]$ and, if we take t'' to be the parent of t' (that is, the parent of t), then $w[t'']$ is a letter which does not appear in $w[t+1:j]$ (which is a strict prefix of the shortest 1-universal factor $w[t+1:t'']$ of w starting on position $t+1$). Note again that, in this case, since $\iota(w[i:j]) = x-y$ and $\iota(w[i:t'']) = x-y+1$, we have that $j < t''$.

In both cases, the label of the nodes found on the path from node $i-1$ towards node t', without the label of $i-1$, form the subsequence $m(w[i:j])$, from Definition 2.3. To this subsequence we add either $w[t]$ (if $t \ne t'$) or $w[t'']$ (otherwise), and obtain an absent subsequence of length $\iota(w[i:j])+1$ of $w[i:j]$. This concludes our proof. □

In order to obtain compact representations of all the SAS and MAS of a word w, we need to define a series of additional arrays.

For a word w of length n, we define the array $\mathrm{dist}[\cdot]$ with n-elements, where, for $i \in [1:n]$, $\mathrm{dist}[i] = \min\{|u| \mid u$ is an absent subsequence of minimal length of $w[i:n]$, starting with $w[i]\}$. The intuition behind the array $\mathrm{dist}[\cdot]$, and the way we will use it, is the following. Assume that w has k arches $\mathrm{ar}_w(1), \ldots, \mathrm{ar}_w(k)$, and $i \in \mathrm{ar}_w(\ell)$. Then there exists an SAS of w, denoted by $w[i_1]\ldots w[i_{k+1}]$, which contains position i only if $\mathrm{dist}[i] = k+1-(\ell-1) = k-\ell+2$. Indeed, an SAS contains one position of every arch, so if i is in that SAS, then it should be its ℓth position, and there should be a word u starting on position i, with $|u| = k-\ell+2$, such that u is not a subsequence of $w[i:n]$. Nevertheless, for all positions j of $\mathrm{ar}_w(\ell)$ it holds that $\mathrm{dist}[j] \ge k-\ell+2$.

Example 4.15. For the word $w = 12213.113312.21$, we have $\mathrm{dist}[i] = 2$ for $i \in [9:13]$ (as exemplified by the word $w[i]3$), $\mathrm{dist}[i] = 3$ for $i \in [3:8]$ (as exemplified by the word $w[i]23$), and $\mathrm{dist}[i] = 4$ for $i \in [1:2]$ (as exemplified by the word $w[i]323$). Note that the single shortest absent sequence in this word is 323.

Lemma 4.16. For a word w of length n, we can construct $\mathrm{dist}[\cdot]$ in $O(n)$ time.

Proof:
It is clear that $\mathrm{dist}[n] = 2$. So, from now on, we show how to compute $\mathrm{dist}[i]$ for $i < n$. It is not hard to see that an absent subsequence of minimal length of $w[i:n]$, starting with $w[i]$, can be obtained by concatenating $w[i]$ to an SAS of $w[i+1:n]$. The latter can be computed, e.g., using $\mathrm{sasRange}[i+1:n]$, and its length is $\mathrm{depth}(i)$. We can thus conclude that, $\mathrm{dist}[i] = \mathrm{depth}(i)+1$. So, as A_w can be computed in linear time, and the depth of the nodes of this tree can also be computed in linear time, the conclusion follows. □

Next, we introduce two additional arrays $\text{sortedLast}[\cdot][\cdot]$ and $\text{Leq}[\cdot][\cdot]$ which are crucial in our representation of the shortest absent subsequences of a word. Let w be a word of length n, with arch factorization $w = \text{ar}_w(1)\cdots\text{ar}_w(k)\,\text{r}(w)$. For each $\ell \in [1 : k-1]$, we define L_ℓ to be the set $\{\text{lastInArch}[\ell][a] \mid a \in \Sigma\}$ of the rightmost positions of any letter $a \in \Sigma$ in the ℓth arch of w. Next we filter out all those positions from L_ℓ which cannot be the first letter of an SAS of $\text{ar}_w(l+1)\cdots\text{ar}_w(k)\,\text{r}(w)$ and store the remaining positions in the set $L'_\ell = L_\ell \setminus \{\text{lastInArch}[\ell][a] \mid \text{dist}(\text{firstInArch}[\ell+1][a]) > k-\ell+1\}$. Finally, we define the array $\text{sortedLast}[\ell][\cdot]$, with $|L'_\ell|$ elements, which contains the elements of L'_ℓ sorted in ascending order. For $\ell = k$, we proceed in the same way, except that in the second step we filter out all positions which correspond to letters occurring in $r(w)$. We define L_k to be the set $\{\text{lastInArch}[k][a] \mid a \in \Sigma\}$ and $L'_k = L_k \setminus \{\text{lastInArch}[k][a] \mid a \in \text{alph}(\text{r}(w))\}$. Then, once more, we define the array $\text{sortedLast}[k][\cdot]$, with $|L'_k|$ elements, which contains the elements of L'_k sorted in ascending order.

Moreover, we define for each ℓ an array with $|\text{ar}_w(\ell)|$ elements $\text{Leq}[\ell][\cdot]$ where $\text{Leq}[\ell][i] = \max(\{t \mid \text{sortedLast}[\ell][t] \le i\} \cup \{-\infty\})$. For simplicity, we assume that $\text{Leq}[\ell][\cdot]$ is indexed by the positions of $\text{ar}_w(\ell)$ (i.e., if $\text{ar}_w(\ell) = w[x : y]$ then $\text{Leq}[\ell][\cdot]$ is indexed by the numbers from x to y).

The intuition behind these arrays is the following. Assume that i is a position in $\text{ar}_w(\ell)$. Then, by the greedy strategy described in the proofs of Theorems 3.1 and 3.4, i can be contained in an SAS only if $i = \text{firstInArch}[\ell][w[i]]$. Moreover, we need to have that $\text{dist}[i] = k-\ell+2$, as explained when the intuition behind the array $\text{dist}[\cdot]$ was described. That is, $\text{lastInArch}[l-1][w[i]]$ needs to be in $L'_{\ell-1}$. Finally, an SAS going through i can only continue with a letter a of the arch $\ell+1$ such that $\text{dist}[\text{firstInArch}[\ell+1][a]] = k-\ell+1$, which, moreover, occurs last time in the arch ℓ on a position $\le i$ (otherwise, the SAS would have two letters in arch ℓ, a contradiction). So, a is exactly one of the letters on the positions given by the elements of $\text{sortedLast}[\ell][1 : \text{Leq}[\ell][i]]$.

Example 4.17. For $w = 12213.113312.21$ we have $\text{ar}_w(1) = 12213$, $\text{ar}_w(2) = 113312$, and $L_1 = \{3, 4, 5\}$. From this set, we eliminate positions $\text{lastInArch}[1][1] = 4$ and $\text{lastInArch}[1][3] = 5$, since $\text{dist}[\text{firstInArch}[2][1]] = \text{dist}[6] = 3 > 2-1+1$ and $\text{dist}[\text{firstInArch}[2][3]] = \text{dist}[8] = 3 > 2-1+1$. So $L'_1 = \{3\}$. Thus, $\text{sortedLast}[1][\cdot]$ has only one element, namely 3. Accordingly, $\text{Leq}[1][i] = -\infty$ for $i \in [1 : 2]$ and $\text{Leq}[1][3] = 1$ (corresponding to the element $\text{sortedLast}[1][1] = 3$ of the array $\text{sortedLast}[1][\cdot]$); moreover, $\text{Leq}[1][4] = \text{Leq}[1][5] = 1$. We have $L_2 = \{9, 10, 11\}$. From this set, we once more eliminate two positions $\text{lastInArch}[2][1] = 10$ and $\text{lastInArch}[2][2] = 11$, because $1, 2 \in \text{alph}(\text{r}(w))$. So $L'_2 = \{9\}$. Thus, $\text{sortedLast}[2][\cdot]$ has only one element, namely 9. Accordingly, $\text{Leq}[2][i] = -\infty$ for $i \in [6 : 8]$ and $\text{Leq}[2][9] = \text{Leq}[2][10] = \text{Leq}[2][11] = 1$ (corresponding to the element $\text{sortedLast}[2][1] = 9$ of the array $\text{sortedLast}[2][\cdot]$).

Based on the arrays $\text{sortedLast}[\ell][\cdot]$, for $\ell \le k$, we define the corresponding arrays $\text{lexSmall}[\ell][\cdot]$ with, respectively, $|L'_\ell|$ elements, where $\text{lexSmall}[\ell][i] = t$ if and only if $w[t] = \min\{w[j] \mid j \in \text{sortedLast}[\ell][1 : i]\}$. In other words, $w[\,\text{lexSmall}[\ell][i]\,]$ is the lexicographically smallest letter occurring on the positions of w corresponding to the first i elements of $\text{sortedLast}[\ell][\cdot]$. Finally, we collect in a list startSAS the elements $\text{firstInArch}[1][a]$, for $a \in \Sigma$, such that $\text{dist}[\text{firstInArch}[1][a]] = k+1$. We assume startSAS is ordered such that $\text{firstInArch}[1][a]$ comes before $\text{firstInArch}[1][b]$ in this list

if a is lexicographically smaller than b. Moreover, let `init` be the element of startSAS such that $w[\mathtt{init}]$ is lexicographically smaller than $w[j]$, for all $j \in \text{startSAS} \setminus \{\mathtt{init}\}$; that is `init` is the first element in startSAS.

Lemma 4.18. For a word w of length n, we can construct $\text{sortedLast}[\cdot][\cdot]$, $\text{Leq}[\cdot][\cdot]$, $\text{lexSmall}[\cdot][\cdot]$, and the list startSAS in $O(n)$ time.

Proof:
By Theorem 4.1, we produce the arch factorisation of w in linear time: $w = \text{ar}_w(1) \cdots \text{ar}_w(k) r(w)$. By Lemmas 4.7 and 4.16, we compute in linear time the arrays $\text{lastInArch}[\cdot][\cdot]$ and $\text{firstInArch}[\cdot][\cdot]$ as well as the array $\text{dist}[\cdot]$ and the set of letters $\text{alph}(r(w))$ (represented in a way which allows to test membership to this set in $O(1)$ time, such as, for instance, the array f from the proof of Theorem 4.1).

Then, we can obtain also in linear time the lists L_ℓ and L'_ℓ, for all ℓ, in a direct way: we simply iterate through all the arches, and for each arch $\text{ar}_w(\ell)$ we iterate through the letters a of Σ, and collect in L'_ℓ those positions $\text{lastInArch}[\ell][a]$ which fulfil the requirements in the definition of this set (which can be clearly tested now in $O(1)$ time).

Note that, when computed in this way, L'_ℓ is not sorted. We can, however, sort in linear time $O(n)$ the set $\cup_{\ell' \leq k} L'_\ell$ in increasing order by counting sort (as all values in these sets are between 1 and n), and obtain an array S of positions of w. We then produce the arrays $\text{sortedLast}[\ell][\cdot]$ in order, for ℓ from 1 to k, by selecting from S the subarray containing positions which are contained in the arch $\text{ar}_w(\ell)$. This is done as follows: we traverse S left to right with a pointer p. After we have produced all the arrays $\text{sortedLast}[i][\cdot]$ for $i < \ell$, p points to the leftmost element t of S such that $t > |\text{ar}_w(1) \cdots \text{ar}_w(\ell - 1)|$. We then copy in $\text{sortedLast}[i][\cdot]$ all the elements t' of S which occur to the right of p and fulfil $t' \leq |\text{ar}_w(1) \cdots \text{ar}_w(\ell)|$. We then move p to the element of S found immediately to the right of the last copied element. Clearly, this process takes linear time.

Computing each array $\text{Leq}[\ell][\cdot]$ is done in a relatively similar way. Intuitively, it is also similar to the merging of the sequence of positions of $\text{ar}_w(\ell)$ (considered in increasing order) to the array $\text{sortedLast}[\ell][\cdot]$. We traverse the arch $\text{ar}_w(\ell)$ left to right, keeping track of the largest element of the sorted array $\text{sortedLast}[\ell][\cdot]$ which we have met so far in this traversal. When we are at position i in the arch and $\text{sortedLast}[\ell][j]$ is the largest element of the sorted array $\text{sortedLast}[\ell][\cdot]$ which we have met when considering the positions up to $i - 1$, we check if $i = \text{sortedLast}[\ell][j + 1]$. If yes, we set $\text{Leq}[\ell][i] = j + 1$ and $\text{sortedLast}[\ell][j + 1]$ becomes the largest element of $\text{sortedLast}[\ell][\cdot]$ seen so far; otherwise we set $\text{Leq}[\ell][i] = j$. Clearly, the computation of a single array $\text{Leq}[\ell][\cdot]$ can be implemented in $O(|\text{ar}_w(\ell)|)$ time, so computing all these arrays takes linear time.

Finally, $\text{lexSmall}[\ell][\cdot]$ can be computed in $O(|L'_\ell|)$ time, for each $\ell \leq k$, by a simple left to right traversal of the array $\text{sortedLast}[\ell][\cdot]$. So, the overall time needed to compute these arrays is linear. The same holds for the computation of startSAS. Clearly, this list can be constructed in $O(\sigma)$ time by simply inserting in it the elements $\text{firstInArch}[1][a]$, for $a \in \Sigma$, such that $\text{dist}[\text{firstInArch}[1][a]] = k + 1$, while going through the letters $a \in \Sigma$ in lexicographical order.

This concludes the proof of our statement. □

We now formalize the intuition that $\text{firstInArch}[1][\cdot]$, $\text{sortedLast}[\cdot][\cdot]$, and $\text{Leq}[\cdot][\cdot]$ are a compact representation of all the SAS of a given word w.

Theorem 4.19. Given a word w of length n with k arches $\mathrm{ar}_w(1), \ldots, \mathrm{ar}_w(k)$, as well as the arrays $\mathrm{firstInArch}[\cdot][\cdot]$, $\mathrm{sortedLast}[\cdot][\cdot]$, $\mathrm{Leq}[\cdot][\cdot]$, $\mathrm{lexSmall}[\cdot][\cdot]$, the set $\mathrm{startSAS}$, and the position `init` of w, which can all be computed in linear time, we can perform the following tasks:

1. We can check in $O(k)$ time if a word u of length $k+1$ is an SAS of w.
2. We can compute in $O(k)$ time the lexicographically smallest SAS of w.
3. We can efficiently enumerate (i.e., with polynomial delay) all the SAS of w.

Proof:
Following Definition 2.3, let $w = \mathrm{ar}_w(1) \cdots \mathrm{ar}_w(k) r(w)$.

We observe that the arrays $\mathrm{firstInArch}[\cdot][\cdot]$, $\mathrm{sortedLast}[\cdot][\cdot]$, $\mathrm{Leq}[\cdot][\cdot]$, and the set $\mathrm{startSAS}$ induce a tree structure $\mathcal{T}_w$ for w, called the SAS-tree of w, and defined inductively as follows:

- The root of $\mathcal{T}_w$ is $\bullet$, while all the nodes of $\mathcal{T}_w$ of depth 1 to k are positions of w. The nodes of depth $k+1$ of $\mathcal{T}_w$ are letters of Σ and the children of each node are totally ordered.
- The children of $\bullet$ are the positions of the list $\mathrm{startSAS}$. These are the only nodes of depth 1. Moreover, the children of $\bullet$ are ordered in the same order as the corresponding positions of $\mathrm{startSAS}$.
- For $\ell \in [1 : k-1]$, we define the nodes of depth $\ell+1$ as follows. The children of a node i of depth ℓ are the nodes $\mathrm{firstInArch}[\ell+1][w[j]]$ where $j \in \mathrm{sortedLast}[\ell][1 : \mathrm{Leq}[\ell][i]]$. The children of node i are ordered such that the node corresponding to $\mathrm{firstInArch}[\ell+1][w[j]]$ comes before $\mathrm{firstInArch}[\ell+1][w[j']]$ if and only if j comes before j' in $\mathrm{sortedLast}[\ell][1 : \mathrm{Leq}[\ell][i]]$.
- For $\ell = k$, we define the nodes of depth $k+1$ (the leaves) as follows. The children of a node i from level k are the letters of $\{w[j] \mid j \in \mathrm{sortedLast}[\ell][1 : \mathrm{Leq}[\ell][i]]\}$. The children of i are ordered such that the node corresponding to letter $w[j]$ comes before the node corresponding to node $w[j']$ if and only if j comes before j' in $\mathrm{sortedLast}[\ell][1 : \mathrm{Leq}[\ell][i]$.
- $\mathcal{T}_w$ contains no other nodes than the ones defined above.

We claim that the paths in the tree from $\bullet$ to the leaves correspond exactly to the set of SAS of w.

Let us show first that each SAS u corresponds to a path in the tree $\mathcal{T}_w$ from $\bullet$ to a leaf. From the definition of $\mathrm{SAS}(w)$, we have that $|u| = k+1$. Firstly, it is clear from the definition of $\mathrm{dist}[\cdot]$ and $\mathrm{startSAS}$, that each SAS starts with a position in $\mathrm{startSAS}$. Then, for $i \in [2 : k]$, assume that the letter $u[i-1]$ occurs on position $j \in \mathrm{ar}_w(i-1)$ of w, and j is a node of the tree. Then, $\mathrm{lastInArch}[i-1][u[i]] \leq j$ (or $\mathrm{next}_w(u[i], j)$ would also be in $\mathrm{ar}_w(i-1)$), so $\mathrm{lastInArch}[i-1][u[i]] \in \mathrm{sortedLast}[i-1][1 : \mathrm{Leq}[i-1][j]]$. Thus $\mathrm{firstInArch}[i][u[i]]$ is a child of j. Applying this inductive argument, we get that $u[1 : k]$ is a path in $\mathcal{T}_w$ from $\bullet$ to a node of depth k. Assume now that the letter $u[k]$ occurs on position $j \in \mathrm{ar}_w(k)$ of w, so, as we have seen, j is a node of the tree. Then, $\mathrm{lastInArch}[k][u[k+1]] \leq j$ and $u[k+1] \in \Sigma \setminus \mathrm{alph}(r(w))$. Thus, $u[k+1] \in \{w[t] \mid t \in \mathrm{sortedLast}[k][1 : \mathrm{Leq}[k][j]]\}$. This means that $u[k+1]$ is a child of j.

To show that each path of the tree from $\bullet$ to the leaves encodes an SAS of w is based on the following inductive argument. Firstly, as an induction base, we consider paths containing only two nodes, namely $\bullet, i_1$; if $i_1 \in \text{startSAS}$, then $u[1] = w[i_1]$ is a word of length 1, whose first occurrence in w is on position $p = i_1$ of w such that $\text{dist}[p] = k + 1$. Now, we define the induction hypothesis. Consider $\ell > 1$ and let us assume that if we have a path $\bullet, i_1, i_2, \ldots, i_\ell$ in $\mathcal{T}_w$, then $i_\ell \in \text{ar}_w(\ell)$, $\text{dist}[i_\ell] = k - \ell + 2$, and $u[1 : \ell] = w[i_1] \cdots w[i_\ell]$ is a subsequence of w such that the shortest factor of w which contains $u[1 : \ell]$ is $w[1 : i_\ell]$. Now, in the induction step, we need to show that the same property holds for a path starting with the root $\bullet$ and containing $\ell + 1$ other nodes. So, basically, we consider a path $\bullet, i_1, i_2, \ldots, i_\ell, i_{\ell+1}$ in $\mathcal{T}_w$. At this point we note that, in order to extend a path $\bullet, i_1, i_2, \ldots, i_\ell$ with a node $i_{\ell+1}$, we need to pick $i_{\ell+1}$ to be a child of i_ℓ, and the path $\bullet, i_1, \ldots, i_\ell$ fulfils the induction hypothesis. Therefore, $i_\ell \in \text{ar}_w(\ell)$, $\text{dist}[i_\ell] = k - \ell + 2$, and $u[1 : \ell] = w[i_1] \cdots w[i_\ell]$ is a subsequence of w such that the shortest factor of w which contains $u[1 : \ell]$ is $w[1 : i_\ell]$. Moreover, as said, $i_{\ell+1}$ is a child of the node i_ℓ, which means that there exists some j such that $i_{\ell+1} = \text{firstInArch}[\ell + 1][w[j]]$ where $j \in \text{sortedLast}[\ell][1 : \text{Leq}[\ell][i_\ell]]$. Therefore, we get that $i_{\ell+1} \in \text{ar}_w(\ell + 1)$ and $\text{dist}[i_{\ell+1}] = k - \ell + 1$, Moreover, we also obtain that $u[1 : \ell + 1] = w[i_1] \cdots w[i_\ell]w[i_{\ell+1}]$ is a subsequence of w such that the shortest factor of w which contains $u[1 : \ell + 1]$ is $w[1 : i_{\ell+1}]$ (as $i_{\ell+1}$ is the first position after i_ℓ where $w[i_{\ell+1}]$ occurs. This proves the induction step.

In conclusion, we can apply this inductive argument for $\ell = k$, so if we have a path $\bullet, i_1, i_2, \ldots, i_k$ in $\mathcal{T}_w$, then $i_k \in \text{ar}_w(k)$, $\text{dist}[i_k] = 2$, and $u[1 : k] = w[i_1] \cdots w[i_k]$ is a subsequence of w such that the shortest factor of w which contains $u[1 : k]$ is $w[1 : i_k]$. Now, this path can only be extended by a child of i_k. The children of i_k are the letters a of $\{w[j] \mid j \in \text{sortedLast}[k][1 : \text{Leq}[k][i_k]]\}$. It is thus clear that $u[1 : k]a$ is an absent factor of w of length $k + 1$, thus an SAS.

Based on this observation, we can prove the statement of the theorem. Note that we do not need to construct this (potentially very large) tree: it is compactly represented by the data structures enumerated in the statement.

For (1) we simply have to check whether u is encoded by a path in $\mathcal{T}_w$. As we do not actually construct this tree, we use the following approach. First we compare $m = |u|$ to $k + 1$. If $m \neq k + 1$, then u is not an SAS of w. If $m = k + 1$, we continue as follows. For i from 1 to $m - 1$, we check if $\text{dist}[\text{firstInArch}[i][u[i]]] = k - i + 2$ and $\text{firstInArch}[i][u[i]] \geq \text{lastInArch}[i][u[i + 1]]$ (i.e., if $u[1 : k]$ is a path in the tree). Then, if all these checks returned true, we check if $u[k + 1]$ is in $\{w[j] \mid j \in \text{sortedLast}[k][1 : \text{Leq}[k][t]]\}$, where $t = \text{firstInArch}[k][u[k]]$. If this final check returns true, then u is an SAS of w. Otherwise, u is not an SAS.

The correctness follows from the properties of $\mathcal{T}_w$.

For (2), we compute an SAS u of length $k + 1$. We define $u[1] = w[\texttt{init}]$. For i from 2 to $k + 1$, we define the ith letter of u as follows:

$$u[i] = w[\ \text{lexSmall}[\,i - 1\,][\,|L'_{i-1}|\,]\ ],\ \text{where } |L'_{i-1}| \text{ is the size of } \text{lexSmall}[i - 1][\cdot].$$

Clearly, this algorithm is derived from the argument showing how the paths of $\mathcal{T}_w$ correspond to the SAS of w. When constructing an SAS we can nondeterministically choose a path in $\mathcal{T}_w$. To

construct the lexicographically smallest SAS, we make the choices in the construction of the SAS in a deterministic way. Our algorithm clearly produces an SAS of w. Moreover, at each step, when we have multiple choices to extend the word u by a letter, we greedily choose this letter to be the lexicographically smallest from the possible choices, which clearly leads to the construction of the lexicographically smallest SAS of w.

For (3), we actually need to traverse the tree $\mathcal{T}_w$ and output the absent subsequences encoded by the paths that lead from $\bullet$ to a leaf, with polynomial delay. If $\mathcal{T}_w$ would be explicitly constructed, this would be done using a standard depth-first traversal of the tree $\mathcal{T}_w$, implemented in a recursive manner: each time a leaf is reached, we need to output the word whose letters correspond to the nodes stored in the recursive-calls stack, ending with the letter corresponding to the leaf. In our case, $\mathcal{T}_w$ is not stored explicitly. However, this is not a problem. We still do a recursive depth-first search in the tree $\mathcal{T}_w$ starting in its root $\bullet$. Each time we call the depth-first search for a node, we first produce the ordered list of its children (this can be clearly done in polynomial time, using the definition of the tree), and then we recursively call the depth-first search for each node in this list. When the list is fully processed, we end the respective call. When we call the depth-first search procedure for a leaf (in the following, we will call this a leaf-call, for simplicity), we output the word of length $k + 1$ which corresponds to the contents of the recursive-calls stack, or, in other words, to the path leading from the root to that leaf (and this encodes, indeed, to a SAS).

To compute this algorithm's time complexity, we need to make some observations. Firstly, we note that in each recursive call of the depth-first search, we do at most σ steps, in order to produce the ordered list of children of that node. Then, we do a recursive call for each of them. That is, we use $O(\sigma)$ time to produce the children list, and then do $O(\sigma)$ new calls. Secondly, let us analyse what happens between two consecutive leaf-calls of the search. In the first leaf-call, we first output the word corresponding to the content of the stack (this takes $O(k)$ time) and then return. Once this leaf-call is ended, a series of other previous calls from the recursive-call stack might also end (the ones corresponding to final nodes in list of childrens); clearly, there are at most k such recursive calls that could end. Then, a call is reached, which was made for a node whose list of children was not yet fully processed. A new call is now initiated, for the first unexplored node of that list, and this leads to a new sequence of at most k recursive-calls ending with a leaf-call. So, to do the processing between two consecutive leaf-calls, we need at most $O(k\sigma)$ time. So: our algorithm simulates a depth-first search of the tree $\mathcal{T}_w$ and outputs all its paths with constant delay. By our argument that there is a one-to-one correspondence between the paths of $\mathcal{T}_w$ and the words from $\mathrm{SAS}(w)$, the correctness of the algorithm is clear. Moreover, as the time between two words are output is upper bounded by $O(k\sigma)$, our claim follows. $\square$

The main point of the previous theorem is to define a compact representation of the words of the set $\mathrm{SAS}(w)$. The fact that this set can be, in fact, represented as a tree of depth $O(k)$ and branching factor $O(\sigma)$ allows us to immediately show that a series of algorithmic tasks can be performed efficiently. However, we do not claim that the solution proposed above for the enumeration of the words in $\mathrm{SAS}(w)$ is optimal. It remains an open problem to define faster enumeration algorithms for the elements of this set.

4.2. A compact representation of the MAS of a word

Similar to the previous section we construct a data structure to store, for given $w \in \Sigma^*$, all the elements of $\mathrm{MAS}(w)$. Since 4.1 does not hold for MAS this data structure is larger than the one for SAS. We start by defining an array $\mathrm{sameNext}[\cdot]$ of size n holds, for each given position i, the first position of $w[i]$ in the word $w[i+1:n]$. We formally define this by $\mathrm{sameNext}[i] = \mathrm{next}(w[i], i+1) = \min\{j \mid w[j] = w[i], j > i\}$, while we assume $\mathrm{sameNext}[i] = \infty$ if $w[i] \notin \mathrm{alph}(w[i+1:n])$.

The array $\mathrm{samePrev}[\cdot]$ of size n holds, for a given position i, the last position of $w[i]$ in the word $w[1:i-1]$. We formally define this with $\mathrm{samePrev}[i] = \mathrm{last}(w[i], i-1) = \max\{j \mid w[j] = w[i], j < i\}$, while we assume $\mathrm{samePrev}[i] = -\infty$ if $w[i] \notin \mathrm{alph}(w[1:i-1])$. Using Line 2 (Algorithm 3) to compute $\mathrm{sameNext}[\cdot]$ ($\mathrm{samePrev}[\cdot]$) is not efficient. Nevertheless we can compute both arrays in linear time.

Lemma 4.20. Given a word w with length $|w| = n$, we can compute in $O(n)$ time the arrays $\mathrm{sameNext}[\cdot]$ and $\mathrm{samePrev}[\cdot]$.

Proof:
The word w needs to be traversed only once from left to right and once from right to left while maintaining an array of size $|\Sigma|$ with the last seen occurrence of each character. Since $|\Sigma| \leq n$, the results follow immediately. □

The following theorem is based on well-known search algorithms on (directed acyclic) graphs, see e.g. [58].

Theorem 4.21. For a word w, we can construct in $O(n^2\sigma)$ time data structures allowing us to efficiently perform the following tasks:

1. We can check in $O(m)$ time if a word u of length m is an MAS of w.
2. We can compute in polynomial time the longest MAS of w.
3. We can check in polynomial time for a given length ℓ if there exists an MAS of length ℓ of w.
4. We can efficiently enumerate (with polynomial delay) all the MAS of w.

Proof:
For a word w, the compact representation of $\mathrm{MAS}(w)$ is based on Theorem 3.1. We define a directed acyclic graph $\mathcal{D}_w$ with the nodes $\{(i,j) \mid 0 \leq j < i \leq n\} \cup \{(0,0), f\}$. The edges (represented as arrows $A \to B$ between nodes A and B) are defined as follows:

- We have an edge $(0,0) \to (i,0)$, if there exists $a \in \Sigma$ such that $i = \mathrm{firstInArch}[1][a]$. This edge is labelled with a.
- For $1 \leq j < i < k \leq n$ we have an edge $(i,j) \to (k,i)$, if there exists $a \in \Sigma$ such that $k = \mathrm{next}_w(a, i+1)$ and $a \in \mathrm{alph}(w[j+1:i])$. This edge is labelled with a.
- We have an edge $(i,j) \to f$, if there exists $b \in \mathrm{alph}(w[j+1:i])$ and $b \notin \mathrm{alph}(w[i+1:n])$. This edge is labelled with b.

We claim that the words in $\mathrm{MAS}(w)$ correspond exactly to the paths in the graph $\mathcal{D}_w$ from $(0,0)$ to f.

Let v be an MAS of w, with $|v| = m$. By Theorem 3.1, there exist positions $0 = i_0 < i_1 < \ldots < i_m < i_{m+1} = n+1$ such that all of the following conditions are satisfied:

(i) $v = w[i_1]\cdots w[i_m]v[m+1]$

(ii) $v[1] \notin \mathrm{alph}(w[1:i_1-1])$

(iii) $v[k] \notin \mathrm{alph}(w[i_{k-1}+1:i_k-1])$ for all $k \in [2:m+1]$

(iv) $v[k] \in \mathrm{alph}(w[i_{k-2}+1:i_{k-1}])$ for all $k \in [2:m+1]$

It immediately follows that $i_1 = \mathrm{firstInArch}[1][v[1]]$. So, there is an edge $(0,0) \to (i_1,0)$. Then, clearly, for all $m \geq \ell \geq 2$, $i_\ell = \mathrm{next}_w(v[\ell], i_{\ell-1}+1)$. Moreover, $v[\ell] \in \mathrm{alph}(w[i_{\ell-2}+1:i_{\ell-1}])$. Thus, there is an edge $(i_{\ell-1}, i_{\ell-2}) \to (i_\ell, i_{\ell-1})$. Similarly, there is an edge from (i_m, i_{m-1}) to f.

For the reverse implication, we consider a path in the graph $\mathcal{D}_w$ from $(0,0)$ to f, namely $(0,0)$, $(i_1,0)$, $(i_2,i_1), \ldots, (i_m, i_{m-1}), f$. We claim that the word $v = w[i_1]\cdots w[i_m]b$ is an MAS of w, for some b such that $b \notin \mathrm{alph}\, w[i_m+1:n]$ and $b \in \mathrm{alph}(w[i_{m-1}+1:i_m])$. It is immediate that v fulfils the four conditions in the statement of Theorem 3.1.

Before solving tasks $(1-4)$, we note that $\mathcal{D}_w$ can be constructed in $O(n^2\sigma)$ time, in a straightforward manner. We store this graph by having for each of its nodes a list of incoming edges (note, at this point, that all these edges have the same label). Using Line 2 trivially, we can produce in $O(n^2\sigma)$ an oracle data structure which contains the answers to any query $\mathrm{next}_w(a,i)$, for all $a \in \Sigma$ and $i \leq n$ (note that a more efficient computation of such a data structure is described in Corollary 4.22). Firstly, we define an edge $(0,0) \to (i,0)$ for all i such that $i = \mathrm{firstInArch}[1][a]$ for some $a \in \Sigma$. Then, for all $i \leq n$ and $a \in \Sigma$, we retrieve $k = \mathrm{next}_w(a, i+1)$ and, then, for all $j < i$, we check if $a \in \mathrm{alph}(w[j+1:i])$ (by simply using our next_w-data structure to see if $\mathrm{next}_w(a, j+1) \leq i$); if $a \in \mathrm{alph}(w[j+1:i])$, we have an edge from (i,j) to (k,i) labelled with a. Finally, for all $j < i$ and $b \in \Sigma$, we have an edge $(i,j) \to f$ labelled with b, if $b \in \mathrm{alph}(w[j+1:i])$ (i.e., $\mathrm{next}_w(b, j+1) \leq i$) and $b \notin \mathrm{alph}(w[i+1:n])$ (i.e., $\mathrm{next}_w(b, i+1) \leq n$ does not hold).

Once $\mathcal{D}_w$ constructed, the correspondence between the words of $\mathrm{MAS}(w)$ and the paths from $(0,0)$ to f in this graph gives us a way to solve all the problems mentioned in the statement of the theorem efficiently. For (2), we simply need to identify the longest such path in $\mathcal{D}_w$, which can be solved in polynomial time in the size of the graph, by a folklore algorithm (see, e.g., [58]). Then, for (3) we need to check if there exists such a path of length ℓ; this can also be done trivially in polynomial time in the size of $\mathcal{D}_w$, by maintaining, for $h = 1$ to ℓ, the set of nodes at distance h from (0.0). Finally, for (4), we need to enumerate efficiently all paths in the graph $\mathcal{D}_w$ from $(0,0)$ to f. This can be implemented in polynomial time, see [59] and the references therein.

We only explain how (1) is performed. Firstly, we can store $\mathcal{D}_w$ as a $(n+1) \times (n+1) \times \sigma$ three dimensional array $D[\cdot][\cdot][\cdot]$, where $D[i][j][a] = k$, for $i, j \in [0:n]$ and $a \in \Sigma$, if and only if there exists an edge from (i,j) to (k,i), and $w[k] = a$. In a sense, in this way we can see the graph $\mathcal{D}_w$ as a DAG whose edges are labelled with letters of Σ. Now, to solve (1), we need to check whether

there exists a path in $\mathcal{D}_w$ from $(0,0)$ to f, labelled with the word u. This can be clearly done in $O(m)$ time. $\square$

Note that the main point of the theorem above is to introduce the graph $\mathcal{D}_w$ as a compact representation of $\mathrm{MAS}(w)$, and to show the correspondence between the strings in $\mathrm{MAS}(w)$ and the paths from $(0,0)$ to f in this graph. Once this correspondence established, we were only interested in establishing that the tasks $(2-4)$ can be solved in polynomial time, and this follows directly from general results regarding directed acyclic graphs. We expect that obtaining more efficient algorithms for the respective problems for the tasks $(2-4)$ from the theorem above would require a deeper understanding of the structure of $\mathcal{D}_w$, and leave this as an open problem.

The last result of this paper shows how to preprocess a word w in order to be able to answer efficiently queries of the following form: for a given u, that is a subsequence of w, which is the shortest string we can append to u in order to obtain an MAS of w?

Corollary 4.22. For a word w of length n, we can construct in $O(n\sigma)$ time data structures allowing us to answer $\mathrm{masExt}(u)$ queries: for a subsequence u of w, decide whether there exists an MAS uv of w, and, if yes, construct such an MAS uv of minimal length. The time needed to answer a query is $O(|v|+|u|)$.

Proof:
Before we begin to show this theorem, we recall an important data structure allowing us to answer range maximum queries. Let A be an array with n elements from a well-ordered set. We define *range maximum queries* RMQ_A for the array of A: $\mathrm{RMQ}_A(i,j) = \mathrm{argmax}\{A[t] \mid t \in [i:j]\}$, for $i,j \in [1:n]$. That is, $\mathrm{RMQ}_A(i,j)$ is the position of the largest element in the subarray $A[i:j]$; if there are multiple positions containing this largest element, $\mathrm{RMQ}_A(i,j)$ is the leftmost of them. (When it is clear from the context, we drop the subscript A).

We will use the following result from [60].

Lemma 4.23. 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 maximum queries* $\mathrm{RMQ}_A(i,j)$, for any $i,j \in [1:n]$.

We now start the actual proof of our theorem and first describe the preprocessing phase.

We first use Lemma 4.20 to compute the arrays $\mathrm{sameNext}$ and $\mathrm{samePrev}$.

We define an $n \times \sigma$ matrix $X[\cdot][\cdot]$, where $X[i][a] = \mathrm{next}_w(a, i+1)$, for all $i \in [0:n-1]$ and $a \in \Sigma$. This matrix can be computed in $O(n\sigma)$ time as follows. We go with i from 1 to n, and we set $X[j][w[i]] = i$ for all $j \in [\mathrm{samePrev}[i] : i-1] \cap [1:n]$. Clearly we can compute an array of size n which contains, for all $i \in [1:n]$, the interval $[\mathrm{samePrev}[i] : i-1] \cap [1:n]$ in linear time.

We define an $n \times \sigma$ matrix $Y[\cdot][\cdot]$, where $Y[i][a] = \mathrm{last}_w(a, i-1)$, for all $i \in [2:n+1]$ and $a \in \Sigma$. This matrix can be computed in $O(n\sigma)$ time trivially, as it equals to the matrix X computed for the mirror image of the word w.

We also preprocess the array $\mathrm{sameNext}[\cdot]$ as in Lemma 4.23, so that we can answer in constant time $\mathrm{RMQ}_{\mathrm{sameNext}}$-queries.

To answer a query $\text{masExt}(u)$, we proceed as follows. If $|u| = 0$, we choose a letter a not occurring in $r(w)$ and simply return $m(w)a$ (where $m(w)$ denotes the concatenation of the last letters of the arches of w) (see the proof of Lemma 4.4), which is clearly an SAS of w. If $|u| > 0$, we run Algorithms 11 and 12, and return the word v computed by the latter.

The correctness of the approach above is explained in the following. For the rest of this proof, assume thus that $|u| > 0$ and that u is a subsequence of w.

First, we run Algorithm 11, which decides whether u can be the prefix of an MAS of w or not. This is an extended and modified version of Algorithm 4, in which we also check the conditions from Theorem 3.1. Note again that we apply this algorithm under the assumption that u is a subsequence of w.

Algorithm 11: isMASPrefix(w,u)

```
   Input: Word w with |w| = n, word u with |u| = m to be tested
 1 j0 ← 0; j1 ← X[0][u[1]]; i ← 2; flag← true;
 2 while i ≤ m and flag == true do
 3 |   j2 ← X[j1][u[i]];
 4 |   j3 ← Y[j2][u[i]];
 5 |   if j3 ∉ [j0 + 1 : j1] then
 6 |   |   flag ← false;
 7 |   else
 8 |   |   j0 ← j1; j1 ← j2; i ← i + 1;
 9 return flag == true ? (j0, j1) : ↑;
```

If Algorithm 11 returns $\uparrow$, then u cannot be the prefix of an MAS of w, as it does not fulfil the conditions of Theorem 3.1.

If Algorithm 11 returns a pair (j_0, j_1), with $j_0 < j_1$, then u can be extended to an MAS of w, $w[1 : j_0]$ is the shortest prefix of w which contains $u[1 : m-1]$ and $w[1 : j_1]$ is the shortest prefix of w which contains $u[1 : m]$. In particular, $u[m-1] = w[j_0]$ and $u[m] = w[j_1]$. We continue with the identification of the shortest string v which we can append to u, based on Theorem 3.1, so that uv becomes an MAS of w. We achieve this by Algorithm 12.

The idea of the algorithm is the following. In order to extend u to a MAS of w, by Theorem 3.1, we need to find a position j between $i_0 + 1$ and i_1, such that $\text{sameNext}[j] > i_1$ (note that such a j always exists, as $\text{sameNext}[i_1] > i_1$). Once j is found, we can safely extend u with $w[j]$, and repeat the procedure, with i_0 and i_1 updated so that they point to the positions of w where the last two letters of the extended u occur. To obtain the shortest word v which extends u to a MAS our algorithm proceeds in a greedy manner, by choosing j such that $\text{sameNext}[j]$ is the largest in the subarray $\text{sameNext}[i_0 + 1 : i_1]$. We will show that this is correct.

So, let us prove that our algorithm works correctly. Firstly, by the remarks we have already made, it is not hard to note that the string v returned by Algorithm 12 has the property that uv fulfils the requirements from Theorem 3.1, so uv is an MAS. Now, for the correctness of the greedy part of the algorithm, we need to show that there is no other strictly shorter string x such that ux is an MAS.

Algorithm 12: masExtend(w, j_0, j_1)

```
  Input: Word w with |w| = n, two positions j0 < j1 of w
1 i0 ← j0; i1 ← j1; i ← 1; flag ← true;
2 while flag == true do
3 |  i2 ← RMQ_sameNext(i0 + 1, i1);
4 |  if sameNext[i2] > n then
5 |  |  v[i] ← w[i2]; flag ← false;
6 |  else
7 |  |  v[i] ← w[i2]; i0 ← i1; i1 ← sameNext[i2]; i ← i + 1;
8 return v;
```

Assume, for the sake of the contradiction, that there exists a string x such that $|x| < |v|$ and ux is an MAS. Let us choose x such that ux is an MAS, whose length is minimal among all MAS which start with u, and, among all such MAS of minimal length, it shares the longest prefix with uv.

As $|u| > 0$ and u itself is not an MAS we have that, $|uv| = h > |ux| = \ell \geq 2$. Let $u_1 = uv$ and $u_2 = ux$. Let $0 = i_0 < i_1 < \ldots < i_{h-1}$ be positions of w such that $w[i_j] = u_1[j]$ for $j \in [1 : h-1]$, as identified greedily by Algorithm 4. Let $0 = g_0 < g_1 < \ldots < g_{\ell-1}$ be positions of w such that $w[g_\ell] = u_2[j]$ for $j \in [1 : \ell - 1]$, as identified by Algorithm 4. First, let us note that there exists $f \leq \ell - 1$ such that $g_f \neq i_f$. Otherwise, $u_2[1 : \ell - 1]$ would be a prefix of u_1, and we would have a position j in $[g_{\ell-1} : g_\ell]$ such that $\mathrm{sameNext}[j] = \infty$. So our algorithm would have also returned a word of length ℓ instead of v, a contradiction. Hence, we can choose the smallest f such that $g_f \neq i_f$, and we know that $f \leq \ell - 1$. By the way we choose i_f in our algorithm, we have that $g_f < i_f$. Let d be minimum such that $g_{f+d} > i_f$; clearly, $d \geq 1$. We can then replace the sequence of positions where the letters of x appear in w by the sequence $g_1 < \ldots < g_{f-1} < i_f < g_{f+d} < \ldots < g_{\ell-1}$ and we note that these positions also fulfil the requirements of Theorem 3.1. The word $uw[g_1] \cdots w[g_{f-1}]w[i_f]w[g_{f+d}]...w[g_{\ell-1}]x[\ell]$ is therefore an MAS, which is either shorter than ux, or has the same length as ux but shares a longer prefix with uv. This is a contradiction with the choice of x.

So, our assumption was false, and it follows that the greedy approach used by Algorithm 12 produces a word v of minimal length such that uv is an MAS. Moreover, the complexity of answering a query is $O(|u| + |v|)$. □

It is worth noting that the lexicographically smallest MAS of a word w is $a^{|w|_a+1}$ where a is the lexicographically smallest letter of Σ which occurs in w.

Acknowledgements:

This is an extended version of the conference paper [61]. We thank the anonymous referees, both of the conference and the journal version of this work, for their valuable comments and suggestions. Last but not least, we thank Tina Ringleb and Maximilian Winkler for carefully proof-reading this paper.

References

[1] Sakarovitch J, Simon I. Subwords. In: Lothaire M (ed.), Combinatorics on Words, chapter 6, pp. 105–142. Cambridge University Press, 1997.

[2] Halfon S, Schnoebelen P, Zetzsche G. Decidability, complexity, and expressiveness of first-order logic over the subword ordering. In: Proc. LICS 2017. 2017 pp. 1–12. doi:10.48550/arXiv.1701.07470.

[3] Karandikar P, Kufleitner M, Schnoebelen P. On the index of Simon's congruence for piecewise testability. *Inf. Process. Lett.*, 2015. **115**(4):515–519. doi:10.1016/j.ipl.2014.11.008.

[4] Karandikar P, Schnoebelen P. The Height of Piecewise-Testable Languages with Applications in Logical Complexity. In: Proc. CSL 2016, volume 62 of *LIPIcs*. 2016 pp. 37:1–37:22.

[5] Karandikar P, Schnoebelen P. The height of piecewise-testable languages and the complexity of the logic of subwords. *Log. Methods Comput. Sci.*, 2019. **15**(2). doi:10.23638/LMCS-15(2:6)2019.

[6] Kuske D. The Subtrace Order and Counting First-Order Logic. In: Proc. CSR 2020, volume 12159 of *Lecture Notes in Computer Science*. 2020 pp. 289–302. doi:10.1007/978-3-030-50026-9_21.

[7] Kuske D, Zetzsche G. Languages Ordered by the Subword Order. In: Proc. FOSSACS 2019, volume 11425 of *Lecture Notes in Computer Science*. 2019 pp. 348–364. doi:10.1007/978-3-030-17127-8_20.

[8] Simon I. Hierarchies of events with dot-depth one - Ph.D. thesis. University of Waterloo, 1972.

[9] Simon I. Piecewise testable events. In: Autom. Theor. Form. Lang., 2nd GI Conf., volume 33 of *LNCS*. 1975 pp. 214–222.

[10] Zetzsche G. The Complexity of Downward Closure Comparisons. In: Proc. ICALP 2016, volume 55 of *LIPIcs*. 2016 pp. 123:1–123:14. doi:10.48550/arXiv.1605.03149.

[11] Freydenberger DD, 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. 2015 pp. 239–248. Available in CoRR. doi:10.48550/arXiv.1509.00622.

[12] Lejeune M, Leroy J, Rigo M. Computing the k-binomial Complexity of the Thue-Morse Word. In: Proc. DLT 2019, volume 11647 of *Lecture Notes in Computer Science*. 2019 pp. 278–291. doi:10.1016/j.jcta.2020.105284.

[13] Leroy J, Rigo M, Stipulanti M. Generalized Pascal triangle for binomial coefficients of words. *Electron. J. Combin.*, 2017. **24**(1.44):36 pp. doi:10.1016/j.aam.2016.04.006.

[14] Mateescu A, Salomaa A, Yu S. Subword Histories and Parikh Matrices. *J. Comput. Syst. Sci.*, 2004. **68**(1):1–21. doi:10.1016/j.jcss.2003.04.001.

[15] Rigo M, Salimov P. Another generalization of abelian equivalence: Binomial complexity of infinite words. *Theor. Comput. Sci.*, 2015. **601**:47–57. doi:10.1016/j.tcs.2015.07.025.

[16] Salomaa A. Connections Between Subwords and Certain Matrix Mappings. *Theoret. Comput. Sci.*, 2005. **340**(2):188–203. doi:10.1016/j.tcs.2005.03.024.

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

[18] Baeza-Yates RA. Searching Subsequences. *Theor. Comput. Sci.*, 1991. **78**(2):363–376.

[19] Bringmann K, Chaudhury BR. Sketching, Streaming, and Fine-Grained Complexity of (Weighted) LCS. In: Proc. FSTTCS 2018, volume 122 of *LIPIcs*. 2018 pp. 40:1–40:16. doi:10.48550/arXiv.1810.01238.

[20] Bringmann K, Künnemann M. Multivariate Fine-Grained Complexity of Longest Common Subsequence. In: Proc. SODA 2018. 2018 pp. 1216–1235. doi:10.1137/1.9781611975031.79.

[21] Maier D. The Complexity of Some Problems on Subsequences and Supersequences. *J. ACM*, 1978. **25**(2):322–336. doi:10.1145/322063.322075.

[22] Wagner RA, Fischer MJ. The String-to-String Correction Problem. *J. ACM*, 1974. **21**(1):168–173.

[23] Sankoff D, Kruskal J. Time Warps, String Edits, and Macromolecules The Theory and Practice of Sequence Comparison. Cambridge University Press, 2000 (reprinted). Originally published in 1983. ISBN-10:1575862174, 13: 978-1575862170.

[24] Pin J. 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*. 2004 p. 5. doi:10.1007/978-3-540-24698-5_4.

[25] Pin J. The influence of Imre Simon's work in the theory of automata, languages and semigroups. *Semigroup Forum*, 2019. **98**:1–8. doi:10.1007/s00233-019-09999-8.

[26] Crochemore M, Melichar B, Tronícek Z. Directed acyclic subsequence graph - Overview. *J. Discrete Algorithms*, 2003. **1**(3-4):255–280. doi:10.1016/S1570-8667(03)00029-7.

[27] Fleischer L, Kufleitner M. Testing Simon's congruence. In: Proc. MFCS 2018, volume 117 of *LIPIcs*. 2018 pp. 62:1–62:13. doi:10.48550/arXiv.1804.10459.

[28] Hebrard JJ. An algorithm for distinguishing efficiently bit-strings by their subsequences. *Theoretical computer science*, 1991. **82**(1):35–49. doi:10.1016/0304-3975(91)90170-7.

[29] Garel E. Minimal Separators of Two Words. In: Proc. CPM 1993, volume 684 of *Lecture Notes in Computer Science*. 1993 pp. 35–53. doi:10.1007/BFb0029795.

[30] Simon I. Words distinguished by their subwords (extended Abstract). In: Proc. WORDS 2003, volume 27 of *TUCS General Publication*. 2003 pp. 6–13.

[31] Tronícek Z. Common Subsequence Automaton. In: Proc. CIAA 2002 (Revised Papers), volume 2608 of *Lecture Notes in Computer Science*. 2002 pp. 270–275. doi:10.1007/3-540-44977-9_28.

[32] Barker L, Fleischmann P, Harwardt K, Manea F, Nowotka D. Scattered Factor-Universality of Words. In: Jonoska N, Savchuk D (eds.), Proc. DLT 2020, volume 12086 of *Lecture Notes in Computer Science*. 2020 pp. 14–28. doi:10.1007/978-3-030-48516-0_2.

[33] Gawrychowski P, Kosche M, Koß T, Manea F, Siemer S. Efficiently Testing Simon's Congruence. In: Bläser M, Monmege B (eds.), 38th International Symposium on Theoretical Aspects of Computer Science, STACS 2021, March 16-19, 2021, Saarbrücken, Germany (Virtual Conference), volume 187 of *LIPIcs*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021 pp. 34:1–34:18. doi: 10.4230/LIPIcs.STACS.2021.34.

[34] Day JD, Fleischmann P, Kosche M, Koß T, Manea F, Siemer S. The Edit Distance to k-Subsequence Universality. In: Bläser M, Monmege B (eds.), 38th International Symposium on Theoretical Aspects of Computer Science, STACS 2021, March 16-19, 2021, Saarbrücken, Germany (Virtual Conference), volume 187 of *LIPIcs*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021 pp. 25:1–25:19. doi: 10.4230/LIPIcs.STACS.2021.25.

[35] Barton C, Heliou A, Mouchard L, Pissis SP. Linear-time computation of minimal absent words using suffix array. *BMC bioinformatics*, 2014. **15**(1):1–10. doi:10.1186/s12859-014-0388-9.

[36] Chairungsee S, Crochemore M. Using minimal absent words to build phylogeny. *Theoretical Computer Science*, 2012. **450**:109–116. doi:10.1016/j.tcs.2012.04.031.

[37] Charalampopoulos P, Crochemore M, Fici G, Mercaş R, Pissis SP. Alignment-free sequence comparison using absent words. *Information and Computation*, 2018. **262**:57–68. doi:10.1016/j.ic.2018.06.002.

[38] Crochemore M, Mignosi F, Restivo A, Salemi S. Data compression using antidictionaries. *Proceedings of the IEEE*, 2000. **88**(11):1756–1768. doi:10.1109/5.892711.

[39] Pratas D, Silva JM. Persistent minimal sequences of SARS-CoV-2. *Bioinformatics*, 2020. doi:10.1093/bioinformatics/btaa686.

[40] Silva RM, Pratas D, Castro L, Pinho AJ, Ferreira PJ. Three minimal sequences found in Ebola virus genomes and absent from human DNA. *Bioinformatics*, 2015. **31**(15):2421–2425. doi:10.1093/bioinformatics/btv189.

[41] Bernardini G, Marchetti-Spaccamela A, Pissis S, Stougie L, Sweering M. Constructing Strings Avoiding Forbidden Substrings. *to appear, Proc. CPM 2021*, 2021. doi:10.4230/LIPIcs.CPM.2021.9.

[42] Crochemore M, Mignosi F, Restivo A. Automata and forbidden words. *Information Processing Letters*, 1998. **67**(3):111–117. doi:10.1016/S0020-0190(98)00104-5.

[43] Fici G, Gawrychowski P. Minimal absent words in rooted and unrooted trees. In: International Symposium on String Processing and Information Retrieval. Springer, 2019 pp. 152–161. doi:10.1007/978-3-030-32686-9_11.

[44] Fici G, Mignosi F, Restivo A, Sciortino M. Word assembly through minimal forbidden words. *Theoretical Computer Science*, 2006. **359**(1-3):214–230. doi:10.1016/j.tcs.2006.03.006.

[45] Fici G, Restivo A, Rizzo L. Minimal forbidden factors of circular words. *Theoretical Computer Science*, 2019. **792**:144–153. doi:10.1016/j.tcs.2018.05.037.

[46] Nakashima Y, Inenaga S, Bannai H, Takeda M. Minimal Unique Substrings and Minimal Absent Words in a Sliding Window. In: SOFSEM 2020: Theory and Practice of Computer Science: 46th International Conference on Current Trends in Theory and Practice of Informatics, SOFSEM 2020, Limassol, Cyprus, January 20–24, 2020, Proceedings, volume 12011. Springer Nature, 2020 p. 148. doi:10.1007/978-3-030-38919-2_13.

[47] Mignosi F, Restivo A, Sciortino M. Words and forbidden factors. *Theoretical Computer Science*, 2002. **273**(1-2):99–117. doi:10.1016/S0304-3975(00)00436-9.

[48] Ayad LA, Badkobeh G, Fici G, Héliou A, Pissis SP. Constructing antidictionaries in output-sensitive space. In: 2019 Data Compression Conference (DCC). IEEE, 2019 pp. 538–547. doi:10.48550/arXiv.1902.04785.

[49] Badkobeh G, Charalampopoulos P, Pissis S. Internal Shortest Absent Word Queries. *to appear, Proc. CPM 2021*, 2021. doi:10.48550/arXiv.2106.01763.

[50] Barton C, Héliou A, Mouchard L, Pissis SP. Parallelising the computation of minimal absent words. In: Parallel Processing and Applied Mathematics, pp. 243–253. Springer, 2016. doi:10.1007/978-3-319-32152-3_23.

[51] Charalampopoulos P, Crochemore M, Pissis SP. On extended special factors of a word. In: International Symposium on String Processing and Information Retrieval. Springer, 2018 pp. 131–138. doi:10.1007/978-3-030-00479-8_11.

[52] Crochemore M, Héliou A, Kucherov G, Mouchard L, Pissis SP, Ramusat Y. Absent words in a sliding window with applications. *Information and Computation*, 2020. **270**:104461. doi:10.1016/j.ic.2019.104461.

[53] Fujishige Y, Tsujimaru Y, Inenaga S, Bannai H, Takeda M. Computing DAWGs and minimal absent words in linear time for integer alphabets. In: 41st International Symposium on Mathematical Foundations of Computer Science (MFCS 2016). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016. doi:10.4230/LIPICS.MFCS.2016.38.

[54] Kitaev S. Patterns in Permutations and Words. Monographs in Theoretical Computer Science. An EATCS Series. Springer, 2011. ISBN 978-3-642-17332-5. doi:10.1007/978-3-642-17333-2.

[55] Crochemore M, Hancart C, Lecroq T. Algorithms on strings. Cambridge University Press, 2007. ISBN 978-0-521-84899-2.

[56] Bender MA, Farach-Colton M. The Level Ancestor Problem simplified. *Theor. Comput. Sci.*, 2004. **321**(1):5–12. doi:10.1016/j.tcs.2003.05.002.

[57] Ben-Amram AM. The Euler Path to Static Level-Ancestors. *CoRR*, 2009. **abs/0909.1030**. 0909.1030, URL http://arxiv.org/abs/0909.1030.

[58] Sedgewick R, Wayne K. Algorithms, 4th Edition. Addison-Wesley, 2011. ISBN 978-0-321-57351-3.

[59] Wasa K. Enumeration of Enumeration Algorithms. *CoRR*, 2016. **abs/1605.05102**. 1605.05102, URL http://arxiv.org/abs/1605.05102.

[60] Bender MA, Farach-Colton M. The LCA Problem Revisited. In: Gonnet GH, Panario D, Viola A (eds.), LATIN 2000: Theoretical Informatics, 4th Latin American Symposium, Punta del Este, Uruguay, April 10-14, 2000, Proceedings, volume 1776 of *Lecture Notes in Computer Science*. Springer, 2000 pp. 88–94. doi:10.1007/10719839_9.

[61] Kosche M, Koß T, Manea F, Siemer S. Absent Subsequences in Words. In: Bell PC, Totzke P, Potapov I (eds.), Reachability Problems - 15th International Conference, RP 2021, Liverpool, UK, October 25-27, 2021, Proceedings, volume 13035 of *Lecture Notes in Computer Science*. Springer, 2021 pp. 115–131. doi:10.1007/978-3-030-89716-1_8. URL https://doi.org/10.1007/978-3-030-89716-1_8.

[62] Garcia SP, Pinho AJ, Rodrigues JM, Bastos CA, Ferreira PJ. Minimal absent words in prokaryotic and eukaryotic genomes. *PLoS One*, 2011. **6**(1):e16065.

[63] Ota T, Morita H. On the adaptive antidictionary code using minimal forbidden words with constant lengths. In: 2010 International Symposium On Information Theory & Its Applications. IEEE, 2010 pp. 72–77. doi:10.1109/ISITA.2010.5649621.

[64] Droubay X, Justin J, Pirillo G. Episturmian words and some constructions of de Luca and Rauzy. *Theor. Comput. Sci.*, 2001. **255**(1-2):539–553.

[65] de Luca A, Glen A, Zamboni LQ. Rich, Sturmian, and trapezoidal words. *Theor. Comput. Sci.*, 2008. **407**(1-3):569–573. doi:10.1016/j.tcs.2008.06.009.

[66] Ayad LAK, Barton C, Pissis SP. A faster and more accurate heuristic for cyclic edit distance computation. *Pattern Recognit. Lett.*, 2017. **88**:81–87. doi:10.1016/j.patrec.2017.01.018.

[67] Glaister I, Shallit J. A Lower Bound Technique for the Size of Nondeterministic Finite Automata. *Information Processing Letters*, 1996. **59**(2):75–77.

[68] Bernardini G, Chen H, Loukides G, Pisanti N, Pissis SP, Stougie L, Sweering M. String Sanitization Under Edit Distance. In: Gørtz IL, Weimann O (eds.), Proc. CPM 2020, volume 161 of *LIPIcs*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020 pp. 7:1–7:14. doi:10.48550/arXiv.2007.08179.

[69] Larsen KG, van Walderveen F. Near-Optimal Range Reporting Structures for Categorical Data. In: Khanna S (ed.), Proc. SODA 2013. SIAM, 2013 pp. 265–276. doi:10.1137/1.9781611973105.20.

[70] Nekrich Y. Efficient range searching for categorical and plain data. *ACM Trans. Database Syst.*, 2014. **39**(1):9:1–9:21. doi:10.1145/2543924.

[71] Chan TM, Patrascu M. Counting Inversions, Offline Orthogonal Range Counting, and Related Problems. In: Charikar M (ed.), Proc. SODA 2010. SIAM, 2010 pp. 161–173. doi:10.1137/1.9781611973075.15.

[72] Schieber B. Computing a Minimum-Weight k-Link Path in Graphs with the Concave Monge Property. In: Proc. SODA 1995. ACM/SIAM, 1995 pp. 405–411.

[73] Aggarwal A, Schieber B, Tokuyama T. Finding a Minimum-Weight k-Link Path Graphs with the Concae Monge Property and Applications. *Discret. Comput. Geom.*, 1994. **12**:263–280.

[74] Bein WW, Larmore LL, Park JK. The d-Edge Shortest-Path Problem for a Monge Graph. *Technical Report*, 1992. **SAND–92-1724C**:1–10.

[75] Allauzen C, Mohri M. Linear-Space Computation of the Edit-Distance between a String and a Finite Automaton. *CoRR*, 2009. **abs/0904.4686**. `0904.4686`, URL `http://arxiv.org/abs/0904.4686`.

[76] Benedikt M, Puppis G, Riveros C. Bounded repairability of word languages. *J. Comput. Syst. Sci.*, 2013. **79**(8):1302–1321. doi:10.1016/j.jcss.2013.06.001.

[77] Benedikt M, Puppis G, Riveros C. Regular Repair of Specifications. In: Proc. LICS 2011. IEEE Computer Society, 2011 pp. 335–344. doi:10.1109/LICS.2011.43.

[78] Pighizzini G. How Hard Is Computing the Edit Distance? *Inf. Comput.*, 2001. **165**(1):1–13. doi:10.1006/inco.2000.2914.

[79] Chatterjee K, Henzinger TA, Ibsen-Jensen R, Otop J. Edit Distance for Pushdown Automata. In: Proc. ICALP 2015, volume 9135 of *Lecture Notes in Computer Science*. Springer, 2015 pp. 121–133. doi:10.1007/978-3-662-47666-6_10.

[80] Wagner RA. Order-n Correction for Regular Languages. *Commun. ACM*, 1974. **17**(5):265–268. doi:10.1145/360980.360995.

[81] Kosolobov D. Computing runs on a general alphabet. *Inf. Process. Lett.*, 2016. **116**(3):241–244. doi:10.1016/j.ipl.2015.11.016

[82] Hamming RW. Error detecting and error correcting codes. *Bell Syst. Tech. J.*, 1950. **29**(2):147–160.

[83] Kosolobov D. Finding the leftmost critical factorization on unordered alphabet. *Theor. Comput. Sci.*, 2016. **636**:56–65.

[84] Gawrychowski P, Kociumaka T, Rytter W, Walen T. Faster Longest Common Extension Queries in Strings over General Alphabets. In: Proc. CPM 2016,, volume 54 of *LIPIcs*. 2016 pp. 5:1–5:13. doi:10.48550/arXiv.1602.00447.

[85] Kosche M, Koß T, Manea F, Siemer S. Absent Subsequences in Words. *CoRR*, 2021. **to appear**. doi:10.48550/arXiv.2108.13968.

[86] Backurs A, Indyk P. Edit Distance Cannot Be Computed in Strongly Subquadratic Time (Unless SETH is False). *SIAM J. Comput.*, 2018. **47**(3):1087–1097. doi:10.1137/15M1053128.

[87] Masek WJ, Paterson M. A Faster Algorithm Computing String Edit Distances. *J. Comput. Syst. Sci.*, 1980. **20**(1):18–31.

[88] Levenshtein VI. Binary codes capable of correcting deletions, insertions, and reversals. *Sov. Phys. - Dokl.*, 1966. **10**(8):707–710.

[89] Bringmann K, Grandoni F, Saha B, Williams VV. Truly Sub-cubic Algorithms for Language Edit Distance and RNA-Folding via Fast Bounded-Difference Min-Plus Product. In: Proc. FOCS 2016. 2016 pp. 375–384. doi:10.1109/FOCS.2016.48.

[90] Jayaram R, Saha B. Approximating Language Edit Distance Beyond Fast Matrix Multiplication: Ultralinear Grammars Are Where Parsing Becomes Hard! In: Proc. ICALP 2017, volume 80 of *LIPIcs*. 2017 pp. 19:1–19:15. doi:10.4230/LIPIcs.ICALP.2017.19.

[91] Cheon H, Han Y. Computing the Shortest String and the Edit-Distance for Parsing Expression Languages. In: Proc. DLT 2020, volume 12086 of *Lecture Notes in Computer Science*. 2020 pp. 43–54. doi:10.1007/978-3-030-48516-0_4.

[92] Cheon H, Han Y, Ko S, Salomaa K. The Relative Edit-Distance Between Two Input-Driven Languages. In: Proc. DLT 2019, volume 11647 of *Lecture Notes in Computer Science*. 2019 pp. 127–139. doi:10.1007/978-3-030-24886-4_9.

[93] Han Y, Ko S, Salomaa K. The Edit-Distance between a Regular Language and a Context-Free Language. *Int. J. Found. Comput. Sci.*, 2013. **24**(7):1067–1082. doi:10.1142/S0129054113400315.

[94] Hirschberg DS. A Linear Space Algorithm for Computing Maximal Common Subsequences. *Commun. ACM*, 1975. **18**(6):341–343.

[95] Dobkin DP, Lipton RJ. On the Complexity of Computations under Varying Sets of Primitives. *J. Comput. Syst. Sci.*, 1979. **18**(1):86–91.

[96] Gabow HN, Tarjan RE. A Linear-Time Algorithm for a Special Case of Disjoint Set Union. In: Proc. 15th STOC. 1983 pp. 246–251.

[97] Cormen TH, Leiserson CE, Rivest RL, Stein C. Introduction to Algorithms, 3rd Edition. MIT Press, 2009. ISBN 978-0-262-03384-8. URL `http://mitpress.mit.edu/books/introduction-algorithms`.

[98] Imai H, Asano T. Dynamic Segment Intersection Search with Applications. In: Proc. FOCS 1984. 1984 pp. 393–402.

[99] Ehlers T, Manea F, Mercas R, Nowotka D. k-Abelian pattern matching. *J. Discrete Algorithms*, 2015. **34**:37–48. doi:10.1016/j.jda.2015.05.004. URL `https://doi.org/10.1016/j.jda.2015.05.004`.

[100] Willard DE. Log-Logarithmic Worst-Case Range Queries are Possible in Space Theta(N). *Inf. Process. Lett.*, 1983. **17**(2):81–84. doi:10.1016/0020-0190(83)90075-3. URL `https://doi.org/10.1016/0020-0190(83)90075-3`.

[101] Aggarwal A, Klawe MM, Moran S, Shor PW, Wilber RE. Geometric Applications of a Matrix-Searching Algorithm. *Algorithmica*, 1987. **2**:195–208.

[102] Babenko MA, Gawrychowski P, Kociumaka T, Starikovskaya TA. Wavelet Trees Meet Suffix Trees. In: Proc. SODA 2015. 2015 pp. 572–591. doi:10.1137/1.9781611973730.39

[103] JáJá J, Mortensen CW, Shi Q. Space-Efficient and Fast Algorithms for Multidimensional Dominance Reporting and Counting. In: Proc. ISAAC 2004, volume 3341 of *Lecture Notes in Computer Science*. 2004 pp. 558–568. doi:10.1007/978-3-540-30551-4_49.

[104] Aggarwal A, Klawe MM. Applications of generalized matrix searching to geometric algorithms. *Discrete Applied Mathematics*, 1990. **27**(1-2):3–23.

[105] Patrascu M. Lower bounds for 2-dimensional range counting. In: Proc. STOC 2007. 2007 pp. 40–46. doi:10.1145/1250790.1250797.

[106] Elzinga CH, Rahmann S, Wang H. Algorithms for subsequence combinatorics. *Theor. Comput. Sci.*, 2008. **409**(3):394–404. doi:10.1016/j.tcs.2008.08.035.

[107] Holub Š, Saari K. On Highly Palindromic Words. *Descrete Applied Mathematics*, 2009. **157**:953–959. doi:10.1016/j.dam.2008.03.039.

[108] Dress AW, Erdös P. Reconstructing Words from Subwords in Linear Time. *Annals of Combinatorics*, 2004. **8**(4):457–462. doi:10.1007/s00026-004-0232-4.

[109] Berstel J, Karhumäki J. Combinatorics on Words – A Tutorial. *BEATCS: Bulletin of the European Association for Theoretical Computer Science*, 2003. **79**.

[110] Manŭch J. Characterization of a Word by its Subwords. In: Developments in Language Theory. 1999 pp. 210–219.

[111] Rozenberg G, Salomaa A (eds.). Handbook of Formal Languages (3 volumes). Springer, 1997.

[112] Fazekas SZ, Manea F, Mercas R, Shikishima-Tsuji K. The Pseudopalindromic Completion of Regular Languages. *Information and Compution*, 2014. **239**:222–236. doi:10.1016/j.ic.2014.09.001.

[113] Kari L, Mahalingam K. Watson-Crick Palindromes in DNA Computing. *Natural Computing*, 2010. **9**(2):297–316. doi:10.1007/s11047-009-9131-2.

[114] Chen HZQ, Kitaev S, Mütze T, Sun BY. On universal partial words. *Electronic Notes in Discrete Mathematics*, 2017. **61**:231–237. doi:10.1016/j.endm.2017.06.043.

[115] Rampersad N, Shallit J, Xu Z. The Computational Complexity of Universality Problems for Prefixes, Suffixes, Factors, and Subwords of Regular Languages. *Fundam. Inf.*, 2012. **116**(1-4):223–236. doi:10.3233/FI-2012-680.

[116] Krötzsch M, Masopust T, Thomazo M. Complexity of universality and related problems for partially ordered NFAs. *Inf. Comput.*, 2017. **255**:177–192. doi:10.1016/j.ic.2017.06.004.

[117] Holzer M, Kutrib M. Descriptional and computational complexity of finite automata - A survey. *Inf. Comput.*, 2011. **209**(3):456–470. doi:10.1016/j.ic.2010.11.013.

[118] Goeckner B, Groothuis C, Hettle C, Kell B, Kirkpatrick P, Kirsch R, Solava RW. Universal partial words over non-binary alphabets. *Theor. Comput. Sci*, 2018. **713**:56–65. doi:10.1016/j.tcs.2017.12.022.

[119] de Bruijn NG. A combinatorial problem. *Koninklijke Nederlandse Akademie v. Wetenschappen*, 1946. **49**:758–764.

[120] Martin MH. A problem in arrangements. *Bull. Amer. Math. Soc.*, 1934. **40**(12):859–864.

[121] Gawrychowski P, Lange M, Rampersad N, Shallit JO, Szykula M. Existential Length Universality. In: Proc. STACS 2020, volume 154 of *LIPIcs*. 2020 pp. 16:1–16:14. doi:10.4230/LIPIcs.STACS.2020.16.

[122] Day JD, Fleischmann P, Manea F, Nowotka D. k-Spectra of Weakly-c-Balanced Words. In: Proc. DLT 2019, volume 11647 of *Lecture Notes in Computer Science*. 2019 pp. 265–277. doi:10.1007/978-3-030-24886-4_20.

[123] van Emde Boas P. Preserving Order in a Forest in Less Than Logarithmic Time and Linear Space. *Inf. Process. Lett.*, 1977. **6**(3):80–82. doi:10.1016/0020-0190(77)90031-X.