Linear-Time Computation of Generalized Minimal Absent Words for Multiple Strings

Kouta Okabe[1], Takuya Mieno[2], Yuto Nakashima[3], Shunsuke Inenaga[3(✉)], and Hideo Bannai[4]

[1] Department of Information Science and Technology, Kyushu University, Fukuoka, Japan
[2] Department of Computer and Network Engineering, University of Electro-Communications, Tokyo, Japan
tmieno@uec.ac.jp
[3] Department of Informatics, Kyushu University, Fukuoka, Japan
{nakashima.yuto.003,inenaga.shunsuke.380}@m.kyushu-u.ac.jp
[4] M&D Data Science Center, Tokyo Medical and Dental University, Tokyo, Japan
hdbn.dsc@tmd.ac.jp

Abstract. A string w is called a *minimal absent word* (*MAW*) for a string S if w does not occur as a substring in S and all proper substrings of w occur in S. MAWs are well-studied combinatorial string objects that have potential applications in areas including bioinformatics, musicology, and data compression. In this paper, we generalize the notion of MAWs to a set $\mathcal{S} = \{S_1, \ldots, S_k\}$ of multiple strings. We first describe our solution to the case of $k = 2$ strings, and show how to compute the set M of MAWs in optimal $O(n+|\mathsf{M}|)$ time and with $O(n)$ working space, where n denotes the total length of the strings in $\mathcal{S}$. We then move on to the general case of $k > 2$ strings, and show how to compute the set M of MAWs in $O(n\lceil k/\log n\rceil + |\mathsf{M}|)$ time and with $O(n(k + \log n))$ bits of working space, in the word RAM model with machine word size $\omega = \log n$. The latter algorithm runs in optimal $O(n + |\mathsf{M}|)$ time for $k = O(\log n)$.

1 Introduction

A non-empty string w is said to be an *absent word* (a.k.a. *a forbidden word*) for a string S if w is *not* a substring of S. An absent word w for S is said to be a *minimal absent word* (*MAW*) for S if all proper substrings of w occur in S. For instance, for string $S = \mathtt{bbacccbaa}$ over an alphabet $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}\}$, the set $\mathsf{MAW}(S)$ of all MAWs for S is $\{\mathtt{aaa}, \mathtt{bbb}, \mathtt{cccc}, \mathtt{d}, \mathtt{ab}, \mathtt{ca}, \mathtt{bc}, \mathtt{aac}, \mathtt{acb}, \mathtt{cbb}, \mathtt{accb}, \mathtt{cbac}, \mathtt{bbaa}\}$. MAWs are combinatorial string objects, and their interesting mathematical properties have extensively been studied in the literature (see [1,7,16,17,19,23] and references therein). MAWs also enjoy several applications including phylogeny [11], data compression [3,15,18], musical information retrieval [14], and bioinformatics [2,12,22,24].

F. M. Nardini et al. (Eds.): SPIRE 2023, LNCS 14240, pp. 331–344, 2023.
https://doi.org/10.1007/978-3-031-43980-3_27

It is known that the number $|\mathsf{MAW}(S)|$ of MAWs for a string S of length n over an alphabet of size σ is $O(\sigma n)$ and that this bound is tight [17]. Crochemore et al. [17] gave an algorithm that computes $\mathsf{MAW}(S)$ in $O(\sigma n)$ time with $O(n)$ working space. Fujishige et al. [20] showed an improved algorithm for computing $\mathsf{MAW}(S)$ in optimal $O(n + |\mathsf{MAW}(S)|)$ time with $O(n)$ working space, for an input string S of length n over an integer alphabet of polynomial size in n. Both of the two aforementioned algorithms utilize an $O(n)$-size string data structure called the *(directed acyclic word graph) DAWG* [9], which recognizes the set of substrings of S, and can be built in $O(n \log \sigma)$ time for general ordered alphabets [9], and in $O(n)$ time for integer alphabets of polynomial size in n [20]. There also exist other efficient algorithms for computing MAWs with other string data structures such as suffix arrays and Burrows-Wheeler transforms [4,8].

The aim of this paper is to extend the notion of MAWs to a set $\mathcal{S} = \{S_1, \ldots, S_k\}$ of multiple k strings. We are aware of a few related attempts in earlier work: Chairungsee and Crochemore [11] introduced a string similarity measure based on the symmetric difference $\mathsf{MAW}(S_1) \triangle \mathsf{MAW}(S_2)$ of the sets of MAWs for two strings S_1 and S_2 to compare. They introduced a length threshold $\ell \geq 1$, and described an approach for computing $(\mathsf{MAW}(S_1) \triangle \mathsf{MAW}(S_2)) \cap \Sigma^\ell$ with the two following steps: First, the tries of size $O(n\ell)$ each representing the substrings of S_1 and S_2 of length up to ℓ are built, where $n = |S_1| + |S_2|$. Then, two tries each representing $\mathsf{MAW}(S_1) \cap \Sigma^\ell$ and $\mathsf{MAW}(S_2) \cap \Sigma^\ell$ are built, which require $O(n\sigma)$ space. Finally, the length-bounded symmetric difference $(\mathsf{MAW}(S_1) \triangle \mathsf{MAW}(S_2)) \cap \Sigma^\ell$ is computed from $\mathsf{MAW}(S_1) \cap \Sigma^\ell$ and $\mathsf{MAW}(S_2) \cap \Sigma^\ell$, but the authors did not explicitly describe how this computation is done in their method. Overall, their algorithm requires $\Omega(n(\ell + \sigma))$ time and space [11][1]. Charalampapaulose et al. [12] tackled the same problem of computing the symmetric difference $\mathsf{MAW}(S_1) \triangle \mathsf{MAW}(S_1)$ (without length threshold ℓ), and proposed a solution that requires $O(\sigma n)$ time and space. Their method firstly computes $\mathsf{MAW}(S_1)$ and $\mathsf{MAW}(S_2)$ separately, and then removes the elements that are in $\mathsf{MAW}(S_1) \cap \mathsf{MAW}(S_2)$. Charalampopoulos, Crochemore, and Pissis [13] presented how to count the number $|\mathsf{MAW}(S_1) \triangle \mathsf{MAW}(S_2)|$ of elements in the symmetric difference $\mathsf{MAW}(S_1) \triangle \mathsf{MAW}(S_2)$ in $O(n)$ time in the case of integer alphabets of polynomial size in n, by avoiding to list the elements explicitly.

Let $\mathcal{S} = \{S_1, \ldots, S_k\}$ be the input set of k strings, and $\mathbf{B} \in \{0,1\}^k$ be a given bit vector of length k. Our problem is to list (generalized) MAWs w for $\mathcal{S}$ and $\mathbf{B}$ such that $w \in \mathsf{MAW}(S_i)$ for every $\mathbf{B}[i] = 1$, and $w \notin \mathsf{MAW}(S_i)$ for every $\mathbf{B}[i] = 0$. For $k = 2$, the aforementioned problem of computing $\mathsf{MAW}(S_1) \triangle \mathsf{MAW}(S_2)$ is equivalent to solving our problem for $\mathbf{B} = 01$ and $\mathbf{B} = 10$. In Sect. 4 and Sect. 5, we deal with the case with $k = 2$, and present an algorithm running in $O(n + |\mathsf{M}_\mathbf{B}|)$ time with $O(n)$ working space, where $\mathsf{M}_\mathbf{B}$ denotes the set of (generalized) MAWs to output for a given bit vector $\mathbf{B}$ (Theorem 2). This immediately gives us an algorithm for listing the elements of the symmetric

[1] The claimed time bound for computing the trie is $O(n\sigma)$ (Theorem 1 of [11]). It seems that the authors regarded the length threshold ℓ as a constant.

difference $\mathsf{MAW}(S_1) \triangle \mathsf{MAW}(S_2)$ in optimal $O(n + |\mathsf{MAW}(S_1) \triangle \mathsf{MAW}(S_2)|)$ time (Corollary 1). In Sect. 6, we deal with the general case of $k > 2$, and extend our solution for $k = 2$ to the general case. Let n be the total length of the input k strings in $\mathcal{S}$. Our solution for general $k > 2$ works in $O(n\lceil k/\log n \rceil + |\mathsf{M}_{\mathbf{B}}|)$ time with $O(n(k + \log n))$ *bits* of working space on the word RAM model with machine word size $\omega = \log n$. Thus, for $k = O(\log n)$, our algorithm runs in optimal $O(n + |\mathsf{M}_{\mathbf{B}}|)$ time. All the bounds claimed in this paper are valid for linearly sortable alphabets, including integer alphabets of polynomial size in n.

As in the previous work [17,20,21], our key data structure is the DAWG for the input set $\mathcal{S}$ of strings. The best-known algorithm for constructing the DAWG for a set of strings of total length n takes $O(n \log \sigma)$ time [10], thus it can require $O(n \log n)$ time for large alphabets. We describe how the DAWG for a given set $\mathcal{S}$ of strings over an integer alphabet of polynomial size in n can be obtained in optimal $O(n)$ time (Theorem 1), which may be of independent interest.

2 Preliminaries

Strings. Let Σ be an ordered alphabet. An element of Σ is called a character. For characters $a, b \in \Sigma$, we write $a \prec b$ (or equivalently $b \succ a$) if a is lexicographically smaller than b. An element of Σ^* is called a string. The length of a string S is denoted by $|S|$. The empty string ε is the string of length 0. If $S = xyz$, then x, y, and z are called a *prefix*, *substring*, and *suffix* of S, respectively. They are called a *proper prefix*, *proper substring*, and *proper suffix* of S if $x \neq S$, $y \neq S$, and $z \neq S$, respectively. Let $\mathsf{Substr}(S)$ denote the set of substrings of string S. For any $1 \leq i \leq |S|$, the i-th character of S is denoted by $S[i]$. For any $1 \leq i \leq j \leq |S|$, $S[i..j]$ denotes the substring of S starting at i and ending at j. For convenience, let $S[i..j] = \varepsilon$ for $0 \leq j < i \leq |S| + 1$. We say that a string w *occurs* in a string S iff w is a substring of S. Note that by definition the empty string ε is a substring of any string S and hence ε always occurs in S.

For a set $\mathcal{S}$ of strings, let $\|\mathcal{S}\|$ denote the total length of the strings in $\mathcal{S}$, that is, $\|\mathcal{S}\| = \sum_{S \in \mathcal{S}} |S|$. Let $\mathsf{Substr}(\mathcal{S})$ denote the set of substrings of the strings in $\mathcal{S}$, that is, $\mathsf{Substr}(\mathcal{S}) = \left(\bigcup_{S \in \mathcal{S}} \{S[i..j] \mid 1 \leq i \leq j \leq |S|\}\right) \cup \{\varepsilon\}$.

Minimal Absent Words (MAWs). A string w is called an *absent word* for a string S if w does not occur in S. Let $\mathsf{AW}(S) = \Sigma^* \setminus \mathsf{Substr}(S)$ denote the set of absent words for a string S. An absent word $w \in \mathsf{AW}(S)$ for string S is called a *minimal absent word* or *MAW* for S if any proper substring of w occurs in S. We denote by $\mathsf{MAW}(S)$ the set of all MAWs for S. Let $\mathsf{nonMAW}(S) = \mathsf{AW}(S) \setminus \mathsf{MAW}(S)$ be the set of absent words for S which are not MAWs. Note that, for strings w and S, it holds that $w \notin \mathsf{MAW}(S)$ iff $w \in \mathsf{Substr}(S) \cup \mathsf{nonMAW}(S)$.

We extend the aforementioned notion of MAWs to a set $\mathcal{S} = \{S_1, \ldots, S_k\}$ of k strings for $k \geq 1$, as follows: Let $\mathbf{B}$ be a bit-vector of length k, and let $\mathcal{S}_{\mathbf{B}}$ be a subset of $\mathcal{S}$ such that $\mathcal{S}_{\mathbf{B}} = \{S_i \mid \mathbf{B}[i] = 1\}$. Let $\overline{\mathcal{S}_{\mathbf{B}}} = \{S_i \mid \mathbf{B}[i] = 0\} = \mathcal{S} \setminus \mathcal{S}_{\mathbf{B}}$. A string w is said to be a MAW for $\mathcal{S}_{\mathbf{B}}$ if (1) $w \in \bigcap_{S_i \in \mathcal{S}_{\mathbf{B}}} \mathsf{MAW}(S_i)$ and (2) $w \notin \bigcup_{S_i \in \overline{\mathcal{S}_{\mathbf{B}}}} \mathsf{MAW}(S_i)$. Condition (1) implies that w is a MAW for any string in $\mathcal{S}_{\mathbf{B}}$.

Condition (2) implies that w is *not* a MAW for any string in $\overline{\mathcal{S}_{\mathbf{B}}}$, which is equivalent to say that $w \in \bigcap_{S_i \in \overline{\mathcal{S}_{\mathbf{B}}}} (\mathsf{Substr}(S_i) \cup \mathsf{nonMAW}(S_i))$. Let $\mathsf{MAW}(\mathcal{S}_{\mathbf{B}})$ be the set of all MAWs for $\mathcal{S}_{\mathbf{B}}$. Here is some example: For string set $\mathcal{S} = \{\mathtt{abaab}, \mathtt{aacbba}\}$ over the alphabet $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}\}$, $\mathsf{MAW}(\mathcal{S}_{10}) = \{\mathtt{aaba}, \mathtt{bab}, \mathtt{bb}, \mathtt{c}\}$, $\mathsf{MAW}(\mathcal{S}_{01}) = \{\mathtt{ab}, \mathtt{baa}, \mathtt{bac}, \mathtt{bbb}, \mathtt{bc}, \mathtt{ca}, \mathtt{cba}, \mathtt{cc}\}$, and $\mathsf{MAW}(\mathcal{S}_{11}) = \{\mathtt{aaa}, \mathtt{d}\}$.

The problem we consider in this paper is the following:

Problem 1 (MAWs for multiple input strings). Given a set $\mathcal{S} = \{S_1, \ldots, S_k\}$ of k strings over an alphabet Σ and a bit vector $\mathbf{B}$ of length k, compute $\mathsf{MAW}(\mathcal{S}_{\mathbf{B}})$.

3 The DAWG Data Structure

We use the *directed acyclic word graph* (*DAWG*) [9] data structure for a set $\mathcal{S} = \{S_1, \ldots, S_k\}$ of k strings, which is a DFA of size $O(\|\mathcal{S}\|)$ that recognizes all suffixes of the strings in $\mathcal{S}$.

To give a formal definition of $\mathsf{DAWG}(\mathcal{S})$, let $\mathsf{End\_Pos}_{\mathcal{S}}(w)$ denote the set of ending positions of all occurrences of a string w in the strings of $\mathcal{S}$, that is,

$$\mathsf{End\_Pos}_{\mathcal{S}}(w) = \{(i, j) \mid S_i[j - |w| + 1..j] = w, 1 \leq i \leq k, 1 \leq j \leq |S_i|\}.$$

We consider an equivalence relation $\equiv_{\mathcal{S}}$ of strings over Σ w.r.t. $\mathcal{S}$ such that, for any two strings w and u, $w \equiv_{\mathcal{S}} u$ iff $\mathsf{End\_Pos}_{\mathcal{S}}(w) = \mathsf{End\_Pos}_{\mathcal{S}}(u)$. For any string $x \in \Sigma^*$, let $[x]_{\mathcal{S}}$ denote the equivalence class for x w.r.t. $\equiv_{\mathcal{S}}$. All the non-substrings $x \notin \mathsf{Substr}(\mathcal{S})$ form a unique equivalence class, called the *degenerate* class.

Definition 1. *The DAWG of a set $\mathcal{S}$ of strings, denoted $\mathsf{DAWG}(\mathcal{S})$, is an edge-labeled DAG (V, E) such that*

$$V = \{[x]_{\mathcal{S}} \mid x \in \mathsf{Substr}(\mathcal{S})\},$$
$$E = \{([x]_{\mathcal{S}}, b, [xb]_{\mathcal{S}}) \mid x, xb \in \mathsf{Substr}(\mathcal{S}), b \in \Sigma\}.$$

We also define the set L of suffix links *of $\mathsf{DAWG}(\mathcal{S})$ by*

$$L = \{([ax]_{\mathcal{S}}, a, [x]_{\mathcal{S}}) \mid x, ax \in \mathsf{Substr}(\mathcal{S}), a \in \Sigma, [ax]_{\mathcal{S}} \neq [x]_{\mathcal{S}}\}.$$

Namely, two substrings x and y in $\mathsf{Substr}(\mathcal{S})$ are represented by the same node of $\mathsf{DAWG}(\mathcal{S})$ iff the ending positions of x and y in the strings of $\mathcal{S}$ are equal. Note that $\mathsf{DAWG}(\mathcal{S})$ does not contain the node for the degenerate class nor its in-coming edges. This is important for $\mathsf{DAWG}(\mathcal{S})$ to have a total linear number of edges [9], and for our linear-time algorithm for listing all the MAWs for a given query.

For convenience, assume that each string S_i in $\mathcal{S} = \{S_1, \ldots, S_k\}$ terminates with a unique end-marker $\#_i$ which does not occur elsewhere, where $\#_i \neq \#_j$ for $i \neq j$. Then $\mathsf{DAWG}(\mathcal{S})$ has exactly k sink nodes, each of which recognizes all the non-empty suffixes of S_i. For each $1 \leq i \leq k$, the sink that recognizes the suffixes of S_i is labeled by i.

The DAWG for a single string T is the DAWG for a singleton $\{T\}$ and is denoted by $\mathsf{DAWG}(T)$.

The state-of-the-art algorithm that builds $\mathsf{DAWG}(\mathcal{S})$ is Blumer et al.'s online algorithm [9] which runs in $O(n \log \sigma)$ time with $O(n)$ space, where $n = \|\mathcal{S}\|$ is the total length of the strings in $\mathcal{S}$ and σ is the alphabet size. Below we describe a faster construction of $\mathsf{DAWG}(\mathcal{S})$ in the case of integer alphabets:

Theorem 1 (Linear-time DAWG construction for a set of strings). *For a given set $\mathcal{S} = \{S_1, \ldots, S_k\}$ of k strings of total length n over an integer alphabet Σ of polynomial size in n, one can build the edge-sorted $\mathsf{DAWG}(\mathcal{S})$ in $O(n)$ time and space.*

Proof. We first create a concatenated string $T = S_1 \cdots S_k$ of total length n from the strings in $\mathcal{S}$. We build $\mathsf{DAWG}(T)$ for the single string T in $O(n)$ time and space, using the algorithm of Fujishige et al. [20,21], where the out-going edges of every node are lexicographically sorted. Our goal is to convert $G_T = \mathsf{DAWG}(T)$ to $G_{\mathcal{S}} = \mathsf{DAWG}(\mathcal{S})$. For a set P of integer pairs and a pair (a, b) of integers, let $P \oplus (a, b) = \{(p + a, q + b) \mid (p, q) \in P\}$. Our key observation is that, for any substrings $w \in \mathsf{Substr}(\mathcal{S})$ that *do not* contain separators $\#_i$ except for their last positions, it holds that

$$\mathsf{End\_Pos}_{\mathcal{S}}(w) = \mathsf{End\_Pos}_{S_1}(w) \cup \left(\bigcup_{2 \le i \le k} \mathsf{End\_Pos}_{S_i}(w) \oplus (i-1, |S_1 \cdots S_{i-1}|) \right). \quad (1)$$

Equation (1) implies that the substrings w of $T = S_1 \cdots S_k$ which are also substrings of $\mathcal{S}$ are represented by essentially the same nodes in G_T and in $G_{\mathcal{S}}$, meaning that there is an injection from the nodes of $G_{\mathcal{S}}$ to the nodes of G_T.

What is left is how to remove the redundant nodes in G_T which represent the substrings y of T containing a separator $\#_i$ inside, which are thus not substrings of $\mathcal{S}$. Let us call the longest path of G_T that represents T as the *spine*. Since each $\#_i$ occurs exactly once in T, any substrings of T that contain $\#_i$ are represented by the spine of G_T. Thus, we can obtain $G_{\mathcal{S}}$ by removing the redundant nodes from the spine of G_T, but we ensure that for every i the suffixes of S_i ending with $\#_i$ are still represented in the graph. This can be achieved as follows: We process $i = k, \ldots, 2$ in decreasing order. We first split the spine into two parts each spelling out $S_1 \cdots S_{k-1}$ and S_k. We remove the nodes in the S_k part which are not reachable from the source of the modified graph, together with their out-going edges and suffix links. This gives us $\mathsf{DAWG}(\{S_1 \cdots S_{k-1}, S_k\})$. After processing $i = k$, we continue the same process for $i = k - 1$ with the remaining spine that spells out $S_1 \cdots S_{k-1}$. After processing $i = 2$, we obtain $G_{\mathcal{S}} = \mathsf{DAWG}(\mathcal{S})$. See Fig. 1 for an example of our construction. It is trivial that all the redundant nodes can be removed in $O(n)$ time. □

We remark that the order of concatenating the strings in $\mathcal{S}$ does not affect the correctness nor the complexity of our algorithm.

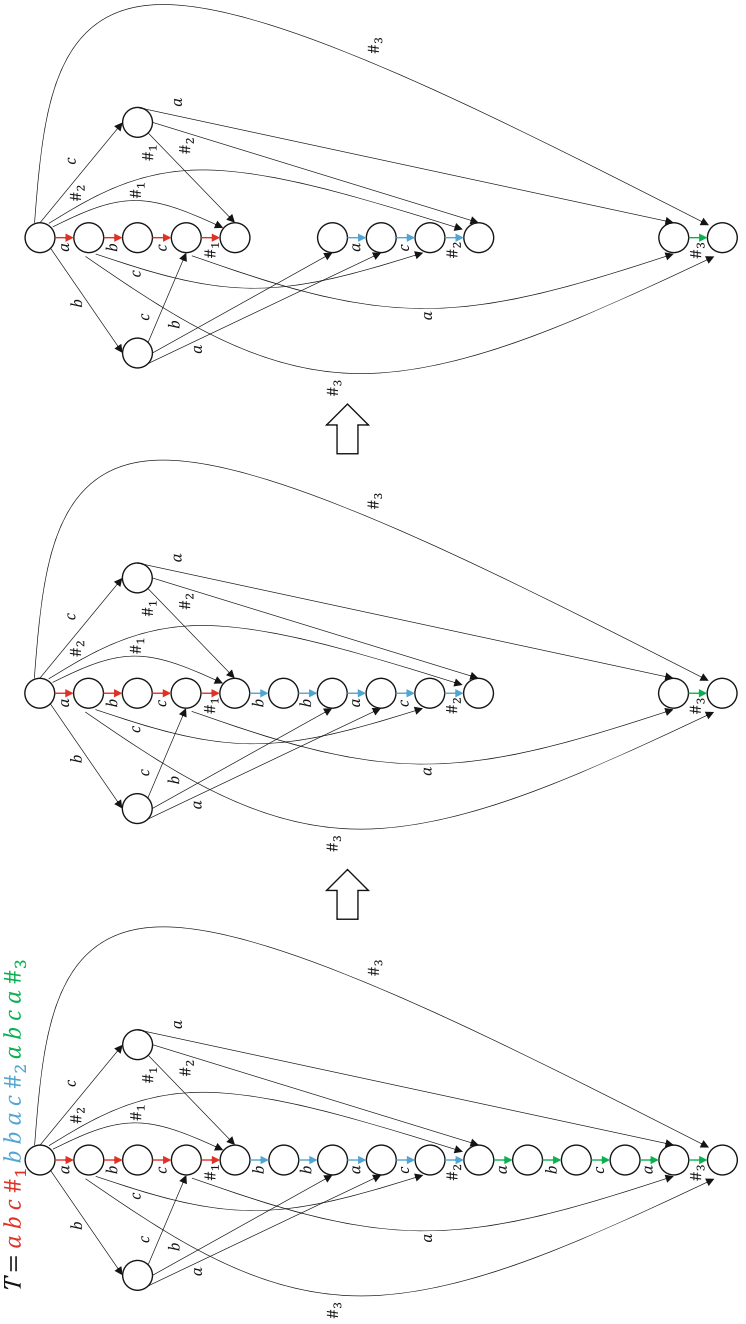


Fig. 1. Illustration for our linear-time construction of $\mathsf{DAWG}(\mathcal{S})$ for a set $\mathcal{S} = \{abc\#_1, bbac\#_2, abca\#_3\}$ of strings. We first build $\mathsf{DAWG}(T)$ for the concatenated string $T = abc\#_1bbac\#_2abca\#_3$. Then, we remove the redundant nodes in the spine of the DAWG for $i = 3$ and then for $i = 2$. This gives us $\mathsf{DAWG}(\mathcal{S})$.

4 Algorithm Overview for $k = 2$

In what follows, we consider the case where our input set $\mathcal{S}$ consists of two strings S_1 and S_2 which respectively terminate with special characters $\#_1$ and $\#_2$. We show how, given a bit vector $\mathbf{B} \in \{00, 01, 10, 11\}$ of length 2, we can compute $\mathsf{MAW}(\mathcal{S}_\mathbf{B})$ in $O(n + |\mathsf{MAW}(\mathcal{S}_\mathbf{B})|)$ time and $O(n)$ working space, where $n = \|\mathcal{S}\|$.

We first build the edge-sorted $\mathsf{DAWG}(\mathcal{S})$ for a given $\mathcal{S} = \{S_1, S_2\}$ in $O(n)$ time and space with Theorem 1. We label each node v of $\mathsf{DAWG}(\mathcal{S})$ by $\#_i$ iff v represents a substring of S_i $(1 \leq i \leq 2)$. Let $\mathsf{label}(v) \in \{\#_1, \#_2, \#_1\#_2\}$ denote the label of node v. The labels of all nodes can be precomputed in $O(n)$ time.

Our algorithm is based on Fujishige et al.'s algorithm [20,21] for computing all the MAWs in the case of a single input string. As such, for each node x of $\mathsf{DAWG}(\mathcal{S})$ we focus on the *shortest* string represented by x and denote it by au, where $a \in \Sigma$ and $u \in \Sigma^*$. We use the suffix link of the node x and its target node y whose *longest* member is u (namely, the first letter a of au is removed by following the suffix link from x to y). For ease of explanation, we identify the node x with the string au, and the node y with the string u.

Fujishige et al.'s algorithm compares the out-going edges of au and those of u one by one in the sorted order. Suppose au has an out-going edge labeled b. If u *does not* have an out-going edge labeled b, then their algorithm outputs aub as a MAW for the input string. Otherwise, it outputs nothing, and the cost is charged to the out-going edge of au labeled b. Each MAW aub in the output is encoded by a tuple (a, i, j) such that $w[i..j] = ub$, thus taking $O(1)$ space. This is how Fujishige et al.'s algorithm works in $O(n + |\mathsf{MAW}(S)|)$ time and with $O(n)$ working space for a single string S.

However, in our case of multiple strings, depending on the label of nodes au, aub and ub, and depending on the value of the given bit vector $\mathbf{B}$, there may exist some edge comparisons that cannot be charged either to the output MAWs or to the out-going edges of node au. It is also possible that even if there is a node representing aub in $\mathsf{DAWG}(\mathcal{S})$, still aub is a MAW for some string(s) in $\mathcal{S}$. To overcome these difficulties, we introduce *skip links* that permit us to avoid unwanted edge character comparisons.

5 Skip Links for $k = 2$

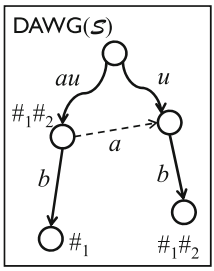
We use the same conventions for the nodes au, aub and u on $\mathsf{DAWG}(\mathcal{S})$ as in the previous section, and also consider the node ub. We have three possible cases for the label of node au, where $\mathsf{label}(au) = \#_1\#_2$, $\mathsf{label}(au) = \#_1$, or $\mathsf{label}(au) = \#_2$. In each of the three cases, there are some sub-cases for the labels of node aub and node ub. By inspection, we obtain all the possible cases that need to be considered, as shown in Fig. 2.

When $\mathbf{B} = 00$, then since $\mathsf{MAW}(\mathcal{S}_{00}) = \Sigma^* \setminus (\mathsf{MAW}(S_1) \cup \mathsf{MAW}(S_2))$, there are no MAWs to output. In what follows, we describe our solutions to the cases with $\mathbf{B} \in \{10, 11\}$. We remark that the case with $\mathbf{B} = 01$ is symmetric to the case with $\mathbf{B} = 10$.

5.1 When B = 10

There are four cases in which we output aub as a MAW for $\mathsf{MAW}(\mathcal{S}_{10})$ (see the table on the left of Fig. 2):

		au		
		$\#_1\#_2$	$\#_1$	$\#_2$
aub	ub	**B**		
$\#_1\#_2$	$\#_1\#_2$	00	-	-
$\#_1$	$\#_1$	00	00	-
	$\#_1\#_2$	01	00	-
$\#_2$	$\#_2$	00	-	00
	$\#_1\#_2$	10	-	00
absent	$\#_1$	10	10	00
	$\#_2$	01	00	01
	$\#_1\#_2$	11	10	01



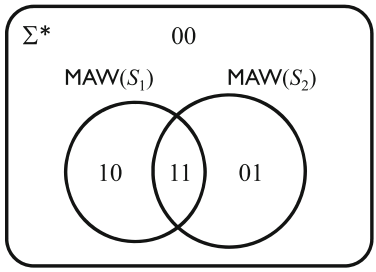


Fig. 2. Left: All possible cases of the labels of the nodes au, aub, and ub, and their corresponding bit vectors $\mathbf{B}$. "absent" refers to the case where there is no out-going edge labeled b from node au. The cells with "-" refer to impossible combinations of node labels. Middle: Illustration for $\mathsf{DAWG}(\mathcal{S})$ which shows the case where au is labeled $\#_1\#_2$, aub is labeled $\#_1$, and ub is labeled $\#_1\#_2$. In this case aub is a MAW in $\mathsf{MAW}(\mathcal{S}_{\mathbf{B}})$ with $\mathbf{B} = 01$ (see the left table). Right: The regions corresponding to the bit vectors $\mathbf{B} \in \{00, 01, 10, 11\}$.

(1) $\mathsf{label}(au) = \#_1\#_2$, $\mathsf{label}(aub) = \#_2$, and $\mathsf{label}(ub) = \#_1\#_2$;
(2) $\mathsf{label}(au) = \#_1\#_2$, $aub \in \mathsf{AW}(\mathcal{S})$, and $\mathsf{label}(ub) = \#_1$;
(3) $\mathsf{label}(au) = \#_1$, $aub \in \mathsf{AW}(\mathcal{S})$, and $\mathsf{label}(ub) = \#_1$;
(4) $\mathsf{label}(au) = \#_1$, $aub \in \mathsf{AW}(\mathcal{S})$, and $\mathsf{label}(ub) = \#_1\#_2$.

When $\mathsf{label}(au) = \#_1\#_2$. We create skip links that simultaneously manage Cases (1) and (2), both having $\mathsf{label}(au) = \#_1\#_2$. We create a selected list $\mathsf{schar}(u)$ of out-going edge labels of node u such that $\mathsf{schar}(u) = \{b \mid \mathsf{label}(ub) = \#_1\}$, where the elements are lexicographically sorted. Let $\mathsf{char}(au)$ be the sorted list of all out-going edge labels of node au. For any list L of characters and any character $c \in \Sigma$, let $\mathsf{succ}(c, L)$ denote the lexicographical successor of c in L. Our algorithm for $\mathbf{B} = 10$ and $\mathsf{label}(au) = \#_1\#_2$ is described in Algorithm 1.

When $\mathsf{label}(au) = \#_1$. We create skip links that simultaneously manage Cases (3) and (4), both having $\mathsf{label}(au) = \#_1$. We create another selected list $\mathsf{schar}'(u)$ of out-going edge labels of node u such that $\mathsf{schar}'(u) = \{b \mid \mathsf{label}(ub) \in \{\#_1, \#_1\#_2\}\}$, where the elements are lexicographically sorted. We use the same $\mathsf{char}(au)$ in the previous case. Our algorithm for $\mathbf{B} = 10$ and $\mathsf{label}(au) = \#_1$ is described in Algorithm 2.

Lemma 1 (Linear-time MAW computation for $\mathbf{B} = 10$). *Given* $\mathbf{B} = 10$, *one can compute* $\mathsf{MAW}(\mathcal{S}_{10})$ *in* $O(n + |\mathsf{MAW}(\mathcal{S}_{10})|)$ *time and* $O(n)$ *working space for integer alphabets of polynomial size in* $n = \|\mathcal{S}\|$.

Algorithm 1: Algorithm for $\mathbf{B} = 10$ and $\mathsf{label}(au) = \#_1\#_2$

Input: A node au of $\mathsf{DAWG}(\mathcal{S})$ such that $\mathsf{label}(au) = \#_1\#_2$, $\mathbf{B} = 10$.
Output: A subset M of MAWs aub with $b \in \Sigma$.

```
1  M ← ∅;
2  U ← char(au) ∪ {$_U} ;                          /* $_U is lex. largest in U */
3  L ← schar(u) ∪ {$_L} ;                  /* $_L is lex. largest in L and $_L ≺ $_u */
4  b̂ ← U[1]; b ← L[1] ;                    /* start with lex. smallest characters */
5  while b ≠ $_L do
6      if b̂ = b then
7          if label(aub) = #_2 and label(ub) = #_1#_2 then
8              M ← M ∪ {aub} ;                                   /* output aub */
9          b̂ ← succ(b̂, U) ;                  /* move to the next character in U */
10         b ← succ(b, L) ;                  /* move to the next character in L */
11     else if b̂ ≻ b then
12         M ← M ∪ {aub} ;                                       /* output aub */
13         b ← succ(b, L) ;                  /* move to the next character in L */
14 return M;
```

Proof. We run Algorithm 1 and Algorithm 2 for every node au of $\mathsf{DAWG}(\mathcal{S})$.

In the preprocessing phase, we build the edge-sorted $\mathsf{DAWG}(\mathcal{S})$ in $O(\|\mathcal{S}\|)$ time and space by Theorem 1. Since the out-going edges of every node are sorted, we can easily compute the sorted lists $\mathsf{char}(au)$, $\mathsf{schar}(u)$, $\mathsf{schar}'(u)$, and $\mathsf{schar}''(u)$ for all nodes in $O(n)$ total time.

Let us consider the complexity of the scanning phase of Algorithm 1. Each edge-label comparison that falls into "$\hat{b} = b$" in line 6 of Algorithm 1 is associated either to the reported MAW aub if $\mathsf{label}(aub) = \#_2$ and $\mathsf{label}(ub) = \#_1\#_2$ (in line 7 and line 8), or to the out-going edge of node au labeled b otherwise. Each edge-label comparison that falls into "$\hat{b} \succ b$" in line 11 is associated to the reported MAW aub in line 12. This ensures the desired time complexity for Algorithm 1. The complexity for Algorithm 2 is similar to show.

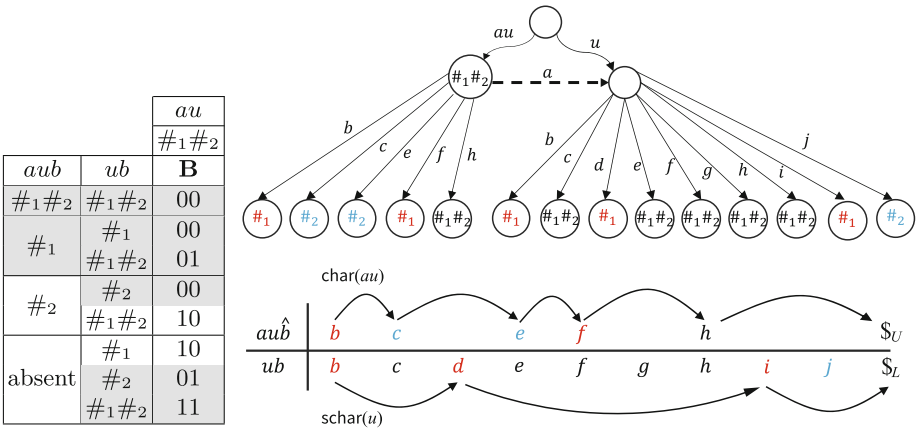
The correctness of Algorithm 1 and Algorithm 2 is immediate from the tables in Fig. 2 and 3. □

5.2 When B = 11

There is a single case in which we output aub as a MAW for $\mathsf{MAW}(\mathcal{S}_{11})$ (see Fig. 2): $\mathsf{label}(au) = \#_1\#_2$, $aub \in \mathsf{AW}(\mathcal{S})$, and $\mathsf{label}(ub) = \#_1\#_2$.

Unwanted comparisons can occur here if $aub \in \mathsf{AW}(\mathcal{S})$, and $\mathsf{label}(ub) = \#_1$ or $\mathsf{label}(ub) = \#_2$. To avoid such comparisons, we consider another carefully selected list $\mathsf{schar}''(u)$ of out-going edge labels of node u such that $\mathsf{schar}''(u) = \{b \mid \mathsf{label}(ub) = \#_1\#_2\}$, where the elements are lexicographically sorted. We can use the same $\mathsf{char}(au)$ in the previous subsection.

We can modify Algorithm 2 for $\mathbf{B} = 01$ with $\mathsf{label}(au) = \#_1$ so that the modified algorithm computes MAWs for $\mathbf{B} = 11$, only by using $\mathsf{schar}''(u)$ in place of $\mathsf{schar}'(u)$. This leads us to the following lemma:



		au
		#1#2
aub	ub	**B**
#1#2	#1#2	00
#1	#1	00
	#1#2	01
#2	#2	00
	#1#2	10
absent	#1	10
	#2	01
	#1#2	11

Fig. 3. Illustration for our algorithm for $\mathbf{B} = 10$ and $\mathsf{label}(au) = \#_1\#_2$. The white cells in the table show the cases where we output elements of $\mathsf{MAW}(\mathcal{S}_{10})$. We compare the labels of the selected out-going edges of node au and u which are connected by the skip links, in sorted order. In this diagram, aud and aui are output in line 12 and auc and aue are output in line 12 of Algorithm 1 as elements of $\mathsf{MAW}(\mathcal{S}_{10})$.

Lemma 2 (Linear-time MAW computation for $\mathbf{B} = 11$). *Given $\mathbf{B} = 11$, one can compute $\mathsf{MAW}(\mathcal{S}_{11})$ in $O(n+|\mathsf{MAW}(\mathcal{S}_{11})|)$ time and $O(n)$ working space for integer alphabets of polynomial size in $n = \|\mathcal{S}\|$.*

5.3 Our Main Result for $k = 2$

Finally we obtain the main result for a case of two strings with $k = 2$.

Theorem 2 (Linear-time MAW computation for a set of two strings). *Given a set $\mathcal{S} = \{S_1, S_2\}$ of two strings of total length n and a bit vector $\mathbf{B} \in \{01, 10, 11\}$, one can compute $\mathsf{MAW}(\mathcal{S}_{\mathbf{B}})$ in $O(n + |\mathsf{MAW}(\mathcal{S}_{\mathbf{B}})|)$ time and $O(n)$ working space for integer alphabets of polynomial size in n.*

The following corollary is immediate from Theorem 2.

Corollary 1. *Given a set $\mathcal{S} = \{S_1, S_2\}$ of two strings of total length n, one can compute $\mathsf{MAW}(S_1) \cap \mathsf{MAW}(S_2)$, $\mathsf{MAW}(S_1) \cup \mathsf{MAW}(S_2)$, and $\mathsf{MAW}(S_1) \triangle \mathsf{MAW}(S_2)$ in $O(n+|\mathsf{MAW}(S_1) \cap \mathsf{MAW}(S_2)|)$ time, $O(n+|\mathsf{MAW}(S_1) \cup \mathsf{MAW}(S_2)|)$ time, and $O(n+|\mathsf{MAW}(S_1) \triangle \mathsf{MAW}(S_2)|)$ time, respectively, using $O(n)$ working space, for integer alphabets of polynomial size in n.*

6 Algorithm for Arbitrary $k > 2$

In this section, we present our algorithm for computing $\mathsf{MAW}(\mathcal{S}_{\mathbf{B}})$ in case where $\mathcal{S} = \{S_1, \ldots, S_k\}$ contains $k > 2$ strings.

Algorithm 2: Algorithm for $\mathbf{B} = 10$ and $\mathsf{label}(au) = \#_1$

Input: A node au of $\mathsf{DAWG}(\mathcal{S})$ such that $\mathsf{label}(au) = \#_1$, $\mathbf{B} = 10$.
Output: A subset M of MAWs aub with $b \in \Sigma$.

```
1  M ← ∅;
2  U ← char(au) ∪ {$_U} ;                    /* $_U is lex. largest in U */
3  L ← schar'(u) ∪ {$_L} ;                   /* $_L is lex. largest in L and $_L ≺ $_u */
4  b̂ ← U[1]; b ← L[1] ;                      /* start with lex. smallest characters */
5  while b ≠ $_L do
6      if b̂ = b then
7          b̂ ← succ(b̂, U) ;                  /* move to the next character in U */
8          b ← succ(b, L) ;                  /* move to the next character in L */
9      else if b̂ ≻ b then
10         M ← M ∪ {aub} ;                   /* output aub */
11         b ← succ(b, L) ;                  /* move to the next character in L */
12 return M;
```

Let $\mathbf{B} \in \{0,1\}^k \setminus \{0^k\}$ be an input bit vector of length $k > 2$. We redefine the labels of the nodes of $\mathsf{DAWG}(\mathcal{S})$ such that $\mathsf{label}(v)[i] = 1$ iff v is a substring of S_i for $1 \le i \le k$. Namely, $\mathsf{label}(v)$ is now also a bit vector of length k.

Let $aub \in \Sigma^*$ ($a, b \in \Sigma$ and $u \in \Sigma^*$) be a candidate of an element of $\mathsf{MAW}(\mathcal{S}_\mathbf{B})$ as in the previous sections, where the suffix link of node au points to node u and node u has an out-going edge labeled b. Then, it follows from the definition of $\mathsf{MAW}(\mathcal{S}_\mathbf{B})$ that $aub \in \mathsf{MAW}(\mathcal{S}_\mathbf{B})$ iff

(A) $\mathsf{label}(aub)[i] = 0$, $\mathsf{label}(au)[i] = 1$, and $\mathsf{label}(ub)[i] = 1$ (i.e. $aub \in \mathsf{MAW}(S_i)$), or
(A') au has no out-going edge labeled b, $\mathsf{label}(au)[i] = 1$, and $\mathsf{label}(ub)[i] = 1$ (i.e. $aub \in \mathsf{MAW}(S_i)$)

for all $1 \le i \le k$ with $\mathbf{B}[i] = 1$, and

(B) $\mathsf{label}(aub)[i] = 1$ (i.e. $aub \in \mathsf{Substr}(S_i)$), or
(C) $\mathsf{label}(aub)[i] = 0$, and $\mathsf{label}(au)[i] = 0$ or $\mathsf{label}(ub)[i] = 0$ (i.e. $aub \in \mathsf{nonMAW}(S_i)$), or
(C') au has no out-going edge labeled b, and $\mathsf{label}(au)[i] = 0$ or $\mathsf{label}(ub)[i] = 0$ (i.e. $aub \in \mathsf{nonMAW}(S_i)$)

for all $1 \le i \le k$ with $\mathbf{B}[i] = 0$.

For each node au in $\mathsf{DAWG}(\mathcal{S})$ whose suffix link points to node u, we create a united single skip link $\mathsf{schar}(ub)$ for the children ub of node u such that $b \in \mathsf{schar}(ub)$ iff $\mathsf{label}(ub)[i] = 1$ for every i with $\mathbf{B}[i] = 1$.

After the above preprocessing is finished, we proceed to the scanning phase of our algorithm. For each node au, we scan the skip links $\mathsf{char}(aub)$ and $\mathsf{schar}(ub)$ in parallel, analogously to the case with $k = 2$. Let $\hat{b} \in \mathsf{char}(aub)$ and $b \in \mathsf{schar}(ub)$. Our algorithm compares these characters in sorted order while keeping the invariant $\hat{b} \succeq b$ as in the case with $k = 2$.

When the comparison falls into the case "$\hat{b} = b$", then we output aub as an element of $\mathsf{MAW}(\mathcal{S}_{\mathbf{B}})$ if Case (A) is satisfied and if Case (B) or Case (C) is satisfied. When the comparison falls into the case "$\hat{b} \succ b$", then we output aub as an element of $\mathsf{MAW}(\mathcal{S}_{\mathbf{B}})$ if Cases (A') and (C') are both satisfied.

This already gives us an $O(nk)$-time algorithm for computing $\mathsf{MAW}(\mathcal{S}_{\mathbf{B}})$ using $O(n(k+\log n))$ *bits* of working space, or alternatively $O(n\lceil k/\log n\rceil)$ words of working space in the word RAM model with machine word size $\omega = \log n$.

We can speed up checking Cases (A), (B), (C) for each node au by using bit masks of size $\omega = \log n$ each stored at nodes aub, au, and ub, from $O(k)$ time to $O(\lceil k/\log n\rceil)$ time. For Cases (A') and (C'), it suffices for us to use only the bit masks stored at nodes au and ub, since node aub does not exist in these cases and we detect this as a result of "$\hat{b} \succ b$" comparison.

Theorem 3 (Efficient MAW computation for a set of k strings). *Given a set $\mathcal{S} = \{S_1, \ldots, S_k\}$ of k strings of total length n and a bit vector $\mathbf{B} \in \{0,1\}^k \setminus \{0^k\}$, one can compute $\mathsf{MAW}(\mathcal{S}_{\mathbf{B}})$ in $O(n\lceil k/\log n\rceil + |\mathsf{MAW}(\mathcal{S}_{\mathbf{B}})|)$ time and $O(n(k+\log n))$ bits of working space (or alternatively $O(n\lceil k/\log n\rceil)$ words of working space), for integer alphabets of polynomial size in n.*

7 Discussions

Béal et al. [6] considered a different version of MAWs $\mathsf{MAW}'(\mathcal{S})$ for a set $\mathcal{S}$ of k strings, where a string $w = aub$ is a MAW for $\mathcal{S} = \{S_1, \ldots, S_k\}$ if $aub \notin \mathsf{Substr}(\mathcal{S})$, $au \in \mathsf{Substr}(S_i)$ and $ub \in \mathsf{Substr}(S_j)$ for some $1 \le i, j \le k$. They gave an $O(\sigma n)$-time and space solution for computing $\mathsf{MAW}'(\mathcal{S})$. This version of MAWs can be computed in optimal $O(n + |\mathsf{MAW}'(\mathcal{S})|)$ time, independently of k, by running our algorithm without skip links. Ayad et al. [3] considered the problem of computing the same version of MAWs of length up to $\ell > 1$.

Independently to our work, the recent work by Béal and Crochemore [5] considered the following problem: Let $\mathcal{T}$ and $\mathcal{R}$ be sets of strings, where $\mathcal{T}$ is called a target and $\mathcal{R}$ is called a reference. A $\mathcal{T}$-specific string with respect to $\mathcal{R}$ is a string u such that $u \in \mathsf{Substr}(\mathcal{T})$, $u \notin \mathsf{Substr}(\mathcal{R})$, $v \in \mathsf{Substr}(\mathcal{R})$ for any proper substring v of u. By definition, a string u is a $\mathcal{T}$-specific string with respect to $\mathcal{R}$ if and only if $u \in \mathsf{MAW}(\mathcal{R}) \cap \mathsf{Substr}(\mathcal{T})$. Béal and Crochemore [5] showed an algorithm for finding all $\mathcal{T}$-specific strings w.r.t. $\mathcal{R}$ in $O(n\sigma)$-time and $O(n)$ space, where n is the total length of the strings in $\mathcal{T}$ and $\mathcal{R}$, assuming that the edges of the DAWG are represented by transition matrices (Proposition 2, [5]). Their algorithm also uses the DAWG built on $\mathcal{T}$ and $\mathcal{R}$ and marks its nodes in an appropriate way (Proposition 1, [5]). This marking technique is very similar to our skip links from Sect. 5 for the case of $k = 2$, and thus our algorithm can be extended to solve this problem in $O(n)$ time and space for integer alphabets.

Acknowledgments. This work was supported by JSPS KAKENHI Grant Numbers JP23H04381 (TM), JP21K17705, JP23H04386 (YN), JP22H03551 (SI), JP20H04141 (HB).

References

1. Akagi, T., et al.: Combinatorics of minimal absent words for a sliding window. Theor. Comput. Sci. **927**, 109–119 (2022). https://doi.org/10.1016/j.tcs.2022.06.002
2. Almirantis, Y., et al.: On avoided words, absent words, and their application to biological sequence analysis. Algorithms Mol. Biol. **12**(1), 5 (2017)
3. Ayad, L.A.K., Badkobeh, G., Fici, G., Héliou, A., Pissis, S.P.: Constructing antidictionaries of long texts in output-sensitive space. Theory Comput. Syst. **65**(5), 777–797 (2021)
4. Barton, C., Heliou, A., Mouchard, L., Pissis, S.P.: Linear-time computation of minimal absent words using suffix array. BMC Bioinform. **15**(1), 388 (2014)
5. Béal, M., Crochemore, M.: Fast detection of specific fragments against a set of sequences. In: Drewes, F., Volkov, M. (eds.) Developments in Language Theory. DLT 2023. LNCS, vol. 13911, pp. 51–60. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-33264-7_5
6. Béal, M., Crochemore, M., Mignosi, F., Restivo, A., Sciortino, M.: Computing forbidden words of regular languages. Fundam. Inform. **56**(1–2), 121–135 (2003)
7. Béal, M.-P., Mignosi, F., Restivo, A.: Minimal forbidden words and symbolic dynamics. In: Puech, C., Reischuk, R. (eds.) STACS 1996. LNCS, vol. 1046, pp. 555–566. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-60922-9_45
8. Belazzougui, D., Cunial, F., Kärkkäinen, J., Mäkinen, V.: Versatile Succinct Representations of the Bidirectional Burrows-Wheeler Transform. In: Bodlaender, H.L., Italiano, G.F. (eds.) ESA 2013. LNCS, vol. 8125, pp. 133–144. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40450-4_12
9. Blumer, A., Blumer, J., Haussler, D., Ehrenfeucht, A., Chen, M.T., Seiferas, J.I.: The smallest automaton recognizing the subwords of a text. Theor. Comput. Sci. **40**, 31–55 (1985)
10. Blumer, A., Blumer, J., Haussler, D., McConnell, R., Ehrenfeucht, A.: Complete inverted files for efficient text retrieval and analysis. J. ACM **34**(3), 578–595 (1987). https://doi.org/10.1145/28869.28873
11. Chairungsee, S., Crochemore, M.: Using minimal absent words to build phylogeny. Theor. Comput. Sci. **450**, 109–116 (2012)
12. Charalampopoulos, P., Crochemore, M., Fici, G., Mercaş, R., Pissis, S.P.: Alignment-free sequence comparison using absent words. Inf. Comput. **262**, 57–68 (2018)
13. Charalampopoulos, P., Crochemore, M., Pissis, S.P.: On extended special factors of a word. In: Gagie, T., Moffat, A., Navarro, G., Cuadros-Vargas, E. (eds.) SPIRE 2018. LNCS, vol. 11147, pp. 131–138. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-00479-8_11
14. Crawford, T., Badkobeh, G., Lewis, D.: Searching page-images of early music scanned with OMR: a scalable solution using minimal absent words. In: ISMIR 2018, pp. 233–239 (2018)
15. Crochemore, M., Mignosi, F., Restivo, A., Salemi, S.: Data compression using antidictionaries. Proc. IEEE **88**(11), 1756–1768 (2000)
16. Crochemore, M., Héliou, A., Kucherov, G., Mouchard, L., Pissis, S.P., Ramusat, Y.: Absent words in a sliding window with applications. Inf. Comput. **270**, 104461 (2020)
17. Crochemore, M., Mignosi, F., Restivo, A.: Automata and forbidden words. Inf. Process. Lett. **67**(3), 111–117 (1998)

18. Crochemore, M., Navarro, G.: Improved antidictionary based compression. In: 12th International Conference of the Chilean Computer Science Society, 2002. Proceedings, pp. 7–13. IEEE (2002)
19. Fici, G.: Minimal forbidden words and applications. Ph.D. thesis, Università di Palermo and Université Paris-Est Marne-la-Vallée (2006)
20. Fujishige, Y., Tsujimaru, Y., Inenaga, S., Bannai, H., Takeda, M.: Computing DAWGs and minimal absent words in linear time for integer alphabets. In: MFCS 2016, vol. 58, pp. 38:1–38:14 (2016)
21. Fujishige, Y., Tsujimaru, Y., Inenaga, S., Bannai, H., Takeda, M.: Linear-time computation of DAWGs, symmetric indexing structures, and MAWs for integer alphabets. Theor. Comput. Sci. (2023, to appear)
22. Koulouras, G., Frith, M.C.: Significant non-existence of sequences in genomes and proteomes. Nucleic Acids Res. **49**(6), 3139–3155 (2021)
23. Mieno, T., et al.: Minimal unique substrings and minimal absent words in a sliding window. In: Chatzigeorgiou, A., et al. (eds.) SOFSEM 2020. LNCS, vol. 12011, pp. 148–160. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-38919-2_13
24. Pratas, D., Silva, J.M.: Persistent minimal sequences of SARS-CoV-2. Bioinformatics **36**(21), 5129–5132 (2020)