Computing DAWGs and Minimal Absent Words in Linear Time for Integer Alphabets

Yuta Fujishige[1], Yuki Tsujimaru[2], Shunsuke Inenaga[3], Hideo Bannai[4], and Masayuki Takeda[5]

1 **Department of Informatics, Kyushu University, Japan**
`yuta.fujishige@inf.kyushu-u.ac.jp`
2 **Department of Electrical Engineering and Computer Science, Kyushu University, Japan**
3 **Department of Informatics, Kyushu University, Japan**
`inenaga@inf.kyushu-u.ac.jp`
4 **Department of Informatics, Kyushu University, Japan**
`bannai@inf.kyushu-u.ac.jp`
5 **Department of Informatics, Kyushu University, Japan**
`takeda@inf.kyushu-u.ac.jp`

Abstract

The *directed acyclic word graph* (*DAWG*) of a string y is the smallest (partial) DFA which recognizes all suffixes of y and has only $O(n)$ nodes and edges. We present the first $O(n)$-time algorithm for computing the DAWG of a given string y of length n over an integer alphabet of polynomial size in n. We also show that a straightforward modification to our DAWG construction algorithm leads to the first $O(n)$-time algorithm for constructing the *affix tree* of a given string y over an integer alphabet. Affix trees are a text indexing structure supporting bidirectional pattern searches. As an application to our $O(n)$-time DAWG construction algorithm, we show that the set $MAW(y)$ of all minimal absent words of y can be computed in *optimal* $O(n + |MAW(y)|)$ time and $O(n)$ working space for integer alphabets.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases string algorithms, DAWGs, suffix trees, minimal absent words

Digital Object Identifier 10.4230/LIPIcs.MFCS.2016.38

1 Introduction

Text indexes are fundamental data structures that allow for efficient processing of string data, and have been extensively studied. Although there are several alternative data structures which can be used as an index, such as suffix trees [18] and suffix arrays [11], in this paper, we focus on *directed acyclic word graphs* (*DAWGs*) proposed by Blumer et al. [3]. Intuitively, the DAWG of string y, denoted $DAWG(y)$, is an edge-labeled DAG obtained by merging isomorphic subtrees of the trie representing all suffixes of string y, called the suffix trie of y. Hence, $DAWG(y)$ can be seen as an automaton recognizing all suffixes of y. Let n be the length of the input string y. Despite the fact that the number of nodes and edges of the suffix trie can be as large as $O(n^2)$, Blumer et al. [3] proved that, surprisingly, $DAWG(y)$ has at most $2n - 1$ nodes and $3n - 4$ edges for $n > 2$. Crochemore [5] showed that $DAWG(y)$ is the smallest (partial) automaton recognizing all suffixes of y, namely, the sub-tree merging operation which transforms the suffix trie to $DAWG(y)$ indeed minimizes the automaton.

Since $DAWG(y)$ is a DAG, in general, more than one string can be represented by its node. It is known that every string represented by the same node of $DAWG(y)$ has the

Table 1 Space requirements and construction times for text indexing structures for input strings of length n over an alphabet of size σ.

	space (in words)	construction time		
		ordered alphabet	integer alphabet	constant alphabet
suffix tries	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n^2)$
suffix trees	$O(n)$	$O(n \log \sigma)$ [12]	$O(n)$ [8]	$O(n)$ [18]
suffix arrays	$O(n)$	$O(n \log \sigma)$ [12]+[11]	$O(n)$ [8]+[11]	$O(n)$ [18]+[11]
DAWGs	$O(n)$	$O(n \log \sigma)$ [3]	$O(n)$ [this work]	$O(n)$ [3]
CDAWGs	$O(n)$	$O(n \log \sigma)$ [4]	$O(n)$ [14]	$O(n)$ [4]
affix trees	$O(n)$	$O(n \log \sigma)$ [10]	$O(n)$ [this work]	$O(n)$ [10]

same set of ending positions in the string y. Due to this property, if z is the longest string represented by a node v of $DAWG(y)$, then any other string represented by the node v is a proper suffix of z. Hence, the *suffix link* of each node of $DAWG(y)$ is well-defined; if ax is the shortest string represented by node v where a is a single character and x is a string, then the suffix link of ax points to the node of $DAWG(y)$ that represents string x.

One of the most intriguing properties of DAWGs is that the suffix links of $DAWG(y)$ for any string y forms the suffix tree [18] of the reversed string of y. Hence, $DAWG(y)$ augmented with suffix links can be seen as a *bidirectional* text indexing data structure. This line of research was followed by other types of bidirectional text indexing data structures such as *symmetric compact DAWGs* (*SCDAWGs*) [4] and *affix trees* [15, 10]. DAWGs with suffix links also have applications to other kinds of string processing problems which are not always easily solvable by using suffix trees or arrays, such as: finding *minimal absent words* for a given string [7, 16], finding *α-gapped repeats* that occur in a given string [17], finding *maximal-exponent repeats* in a given overlap-free string [1], computing the *Lempel-Ziv 77 factorization* [20] of a given string in an online manner and with compact space [19].

Time complexities for constructing text indexing data structures depend on the underlying alphabet. See Table 1. For a given string y of length n over an ordered alphabet of size σ, the suffix tree [12], the suffix array [11], the DAWG, and the *compact DAWGs* (*CDAWGs*) [4] of y can all be constructed in $O(n \log \sigma)$ time. These immediately lead to $O(n)$-time construction algorithms for a constant alphabet.

In this paper, we are particularly interested in input strings of length n over an *integer alphabet* of polynomial size in n. Farach-Colton et al. [8] proposed the first $O(n)$-time suffix tree construction algorithm for integer alphabets. Since the out-edges of every node of the suffix tree constructed by McCreight's [12] and Farach-Colton et al.'s algorithms are lexicographically sorted, and since sorting is an obvious lower-bound for constructing edge-sorted suffix trees, the above-mentioned suffix-tree construction algorithms are optimal for ordered and integer alphabets, respectively. Since the suffix array of y can be easily obtained in $O(n)$ time from the edge-sorted suffix tree of y, suffix arrays can also be constructed in optimal time. In addition, since the edge-sorted suffix tree of y can easily be constructed in $O(n)$ time from the edge-sorted CDAWG of y, and since the edge-sorted CDAWG of y can be constructed in $O(n)$ time from the edge-sorted DAWG of y [4], sorting is also a lower-bound for constructing edge-sorted DAWGs and edge-sorted CDAWGs. Using the technique of Narisawa et al. [14], edge-sorted CDAWGs can be constructed in optimal $O(n)$ time for integer alphabets. On the other hand, the only known algorithm to construct DAWGs was Blumer et al.'s $O(n \log \sigma)$-time online algorithm [3] for ordered alphabets of size σ, which results in $O(n \log n)$-time DAWG construction for integer alphabets.

In this paper, we close the gap between the upper and lower bounds for DAWG construction, by proposing the first $O(n)$-time algorithm to construct edge-sorted DAWGs for integer alphabets. Our algorithm also computes the suffix links, and can thus be applied to various kinds of string processing problems. Our algorithm builds $DAWG(y)$ for a given string y by transforming the suffix tree of y to $DAWG(y)$. In other words, our algorithm simulates the minimization of the suffix trie of y to $DAWG(y)$ using only $O(n)$ time and space.

A simple modification to our $O(n)$-time DAWG construction algorithm also leads us to the first $O(n)$-time algorithm to construct affix trees for integer alphabets. We remark that the previous best known affix-tree construction algorithm of Maaß [10] requires $O(n \log n)$ time for integer alphabets.

As an application of our $O(n)$-time DAWG construction algorithm, we present the first optimal time algorithm to compute *minimal absent words* for a given string. Let $MAW(y)$ be the set of minimal absent words of y. Crochemore et al. [7] proposed an algorithm to compute $MAW(y)$ in $\Theta(n\sigma)$ time and $O(n)$ working space. Their algorithm first constructs $DAWG(y)$ with suffix links in $O(n \log \sigma)$ time and compute $MAW(y)$ in $O(n\sigma)$ time using $DAWG(y)$ and its suffix links. Since $|MAW(y)| = O(n\sigma)$, the output size $|MAW(y)|$ is hidden in the running time of their algorithm. In this paper, we show that $MAW(y)$ can be computed in *output-sensitive* $O(n + |MAW(y)|)$ optimal time for integer alphabets. We first construct edge-sorted $DAWG(y)$ in $O(n)$ time using the algorithm we propose in this paper. Then, we show that a slight modification to Crochemore et al.'s algorithm [7] finds $MAW(y)$ in $O(n + |MAW(y)|)$ time. We emphasize that for non-constant alphabets Crochemore et al.'s algorithm takes super-linear time in terms of the input string length independently of the output size $|MAW(y)|$, and thus our results greatly improves the efficiency for integer alphabets. Belazzougui et al. [2] showed that using a representation of the bidirectional BWT of the input string y, $MAW(y)$ can be computed in $O(n + |MAW(y)|)$ time. However, the construction time for the representation of the bidirectional BWT is not given in [2].

Our result can also be applied to recent work by Crochemore et al. [6] for string comparison with minimal absent words, resulting in a more efficient algorithm for string comparison with minimal absent words for integer alphabets.

2 Preliminaries

2.1 Strings

Let Σ denote the alphabet. An element of Σ^* is called a *string*. Let ε denote the empty string, and let $\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$. For any string y, we denote its length by $|y|$. For any $1 \leq i \leq |y|$, we use $y[i]$ to denote the ith character of y. If $y = uvw$ with $u, v, w \in \Sigma^*$, then u, v, and w are said to be a *prefix*, *substring*, and *suffix* of y, respectively. For any $1 \leq i \leq j \leq |y|$, $y[i..j]$ denotes the substring of y which begins at position i and ends at position j. For convenience, let $y[i..j] = \varepsilon$ if $i > j$. Let $Substr(y)$ and $Suffix(y)$ denote the set of all substrings and that of all suffixes of y, respectively.

Throughout this paper, we will use y to denote the input string. For any string $x \in \Sigma^*$, we define $BegPos(x) = \{i \mid i \in [1, |y| - |x| + 1], y[i..i + |x| - 1] = x\}$, $EndPos(x) = \{i \mid i \in [|x|, |y|], y[i - |x| + 1..i] = x\}$, i.e., the set of beginning and end positions of occurrences of x in y. For any strings u, v, we write $u \equiv_L v$ (resp. $u \equiv_R v$) when $BegPos(u) = BegPos(v)$ (resp. $EndPos(u) = EndPos(v)$). For any string $x \in \Sigma^*$, the equivalence classes with respect to $\equiv_L$ and $\equiv_R$ that x belongs to, are respectively denoted by $[x]_L$ and $[x]_R$. Also, $\overrightarrow{x}$ and $\overleftarrow{x}$ respectively denote the longest elements of $[x]_L$ and $[x]_R$.

For any set S of strings where no two strings are of the same length, let $\mathrm{long}(S) = \arg\max\{|x| \mid x \in S\}$ and $\mathrm{short}(S) = \arg\min\{|x| \mid x \in S\}$.

In this paper, we assume that the input string y of length n is over the integer alphabet $[1, n^c]$ for some constant c, and that the last character of y is a unique character denoted by \$ that does not occur elsewhere in y. Our model of computation is a standard word RAM of machine word size $\log_2 n$. Space complexities will be evaluated by the number of words (not bits).

2.2 Suffix trees and DAWGs

Suffix trees [18] and directed acyclic word graphs (*DAWGs*) [3] are fundamental text data structures. Both of these data structures are based on suffix tries. The *suffix trie* for string y, denoted $\mathit{STrie}(y)$, is a trie representing $\mathit{Substr}(y)$, formally defined as follows.

▶ **Definition 1.** $\mathit{STrie}(y)$ for string y is an edge-labeled rooted tree (V_T, E_T) such that

$$\begin{aligned} V_T &= \{x \mid x \in \mathit{Substr}(y)\} \\ E_T &= \{(x, b, xb) \mid x, xb \in V_T, b \in \Sigma\}. \end{aligned}$$

The second element b of each edge (x, b, xb) is the label of the edge. We also define the set L_T of labeled "reversed" edges called the *suffix links* of $\mathit{STrie}(y)$ by

$$L_T = \{(ax, a, x) \mid x, ax \in \mathit{Substr}(y), a \in \Sigma\}.$$

As can be seen in the above definition, each node v of $\mathit{STrie}(y)$ can be identified with the substring of y that is represented by v. Assuming that string y terminates with a unique character that appears nowhere else in y, for each suffix $y[i..|y|] \in \mathit{Suffix}(y)$ there is a unique leaf ℓ_i in $\mathit{STrie}(y)$ such that the suffix $y[i..|y|]$ is spelled out by the path from the root to ℓ_i.

It is well known that $\mathit{STrie}(y)$ requires $O(n^2)$ space. One idea to reduce its space to $O(n)$ is to contract each path consisting only of non-branching edges into a single edge labeled with a non-empty string. This leads to the suffix tree $\mathit{STree}(y)$ of string y. Following conventions from [4, 9], $\mathit{STree}(y)$ is defined as follows.

▶ **Definition 2.** $\mathit{STree}(y)$ for string y is an edge-labeled rooted tree (V_S, E_S) such that

$$\begin{aligned} V_S &= \{\overrightarrow{x} \mid x \in \mathit{Substr}(y)\} \\ E_S &= \{(x, \beta, x\beta) \mid x, x\beta \in V_S, \beta \in \Sigma^+, b = \beta[1], \overrightarrow{xb} = x\beta\}. \end{aligned}$$

The second element β of each edge $(x, \beta, x\beta)$ is the label of the edge. We also define the set L_S of labeled "reversed" edges called the *suffix links* of $\mathit{STree}(y)$ by

$$L_S = \{(ax, a, x) \mid x, ax \in V_S, a \in \Sigma\},$$

and denote the tree (V_S, L_S) of the suffix links by $\mathit{SLT}(y)$.

Observe that each internal node of $\mathit{STree}(y)$ is a branching internal node in $\mathit{STrie}(y)$. Note that for any $x \in \mathit{Substr}(y)$ the leaves in the subtree rooted at $\overrightarrow{x}$ correspond to $\mathit{BegPos}(x)$. By representing each edge label β with a pair of integers (i, j) such that $y[i..j] = \beta$, $\mathit{STree}(y)$ can be represented with $O(n)$ space.

An alternative way to reduce the size of $\mathit{STrie}(y)$ to $O(n)$ is to regard $\mathit{STrie}(y)$ as a partial DFA which recognizes $\mathit{Suffix}(y)$, and to minimize it. This leads to the directed acyclic word graph $\mathit{DAWG}(y)$ of string y. Following conventions from [4, 9], $\mathit{DAWG}(y)$ is defined as follows.

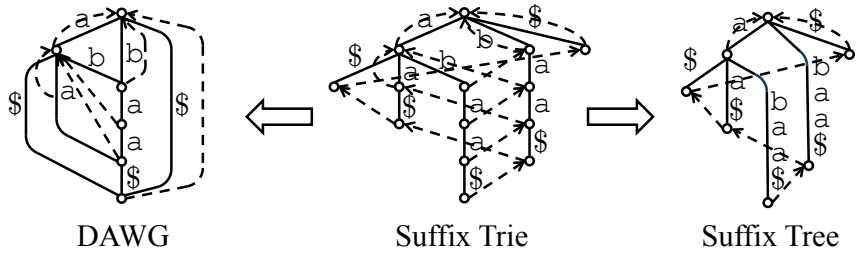

Figure 1 $STrie(y)$, $STree(y)$, and $DAWG(y)$ for string $y = \texttt{abaa\$}$. The solid arcs represent edges, and the broken arcs represent suffix links.

▶ **Definition 3.** $DAWG(y)$ of string y is an edge-labeled DAG (V_D, E_D) such that

$$V_D = \{[x]_R \mid x \in Substr(y)\}$$
$$E_D = \{([x]_R, b, [xb]_R) \mid x, xb \in Substr(y), b \in \Sigma\}.$$

We also define the set L_D of labeled "reversed" edges called the *suffix links* of $DAWG(y)$ by

$$L_D = \{([ax]_R, a, [x]_R) \mid x, ax \in Substr(y), a \in \Sigma, [ax]_R \neq [x]_R\}.$$

See Figure 1 for examples of $STrie(y)$, $STree(y)$, and $DAWG(y)$.

▶ **Theorem 4** ([3]). *For any string y of length $n > 2$, the number of nodes in $DAWG(y)$ is at most $2n - 1$ and the number of edges in $DAWG(y)$ is at most $3n - 4$.*

Minimization of $STrie(y)$ to $DAWG(y)$ can be done by merging isomorphic subtrees of $STrie(y)$ which are rooted at nodes connected by a chain of suffix links of $STrie(y)$. Since the substrings represented by these merged nodes end at the same positions in y, each node of $DAWG(y)$ forms an equivalence class $[x]_R$. We will make an extensive use of this property in our $O(n)$-time construction algorithm for $DAWG(y)$ over an integer alphabet.

2.3 Minimal Absent Words

A string x is said to be an *absent word* of another string y if $x \notin Substr(y)$. An absent word x of y is said to be a *minimal absent word* (MAW) of y if $Substr(x) \setminus \{x\} \subset Substr(y)$. The set of all MAWs of y is denoted by $MAW(y)$. For example, if $\Sigma = \{\texttt{a}, \texttt{b}, \texttt{c}\}$ and $y = \texttt{abaab}$, then $MAW(y) = \{\texttt{aaa}, \texttt{aaba}, \texttt{bab}, \texttt{bb}, \texttt{c}\}$.

▶ **Lemma 5** ([13]). *For any string $y \in \Sigma^*$, $\sigma \leq |MAW(y)| \leq (\sigma_y - 1)(|y| - 1) + \sigma$, where $\sigma = |\Sigma|$ and σ_y is the number of distinct characters occurring in y. This bound is tight.*

The next lemma follows from the definition of MAWs.

▶ **Lemma 6.** *Let y be any string. For any characters $a, b \in \Sigma$ and string $x \in \Sigma^*$, $axb \in MAW(y)$ iff $axb \notin Substr(y)$, $ax \in Substr(y)$, and $xb \in Substr(y)$.*

By Lemma 6, we can encode each MAW axb of y in $O(1)$ space by (i, j, b), where $ax = y[i..j]$.

3 Constructing DAWGs in $O(n)$ Time for Integer Alphabet

In this section, we present an optimal $O(n)$-time algorithm to construct $DAWG(y)$ with suffix links L_D for a given string y of length n over an integer alphabet. Our algorithm constructs $DAWG(y)$ with suffix links L_D from $STree(y)$ with suffix links L_S. The following result is known.

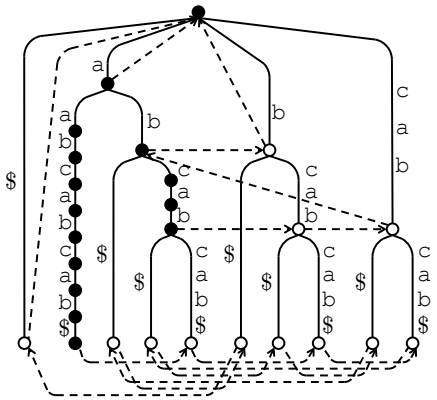

Figure 2 An example of $STree'(y)$ with string $y = \texttt{aabcabcab\$}$.

▶ **Theorem 7** ([8]). *Given a string y of length n over an integer alphabet, edge-sorted $STree(y)$ with suffix links L_S can be computed in $O(n)$ time.*

Let $\mathcal{L}$ and $\mathcal{R}$ be, respectively, the sets of longest elements of all equivalence classes on y w.r.t. $\equiv_L$ and $\equiv_R$, namely, $\mathcal{L} = \{\overrightarrow{x} \mid x \in Substr(y)\}$ and $\mathcal{R} = \{\overleftarrow{x} \mid x \in Substr(y)\}$. Let $STree'(y) = (V'_S, E'_S)$ be the edge-labeled rooted tree obtained by adding extra nodes for strings in $\mathcal{R}$ to $STree(y)$, namely,

$$\begin{aligned} V'_S &= \{x \mid x \in \mathcal{L} \cup \mathcal{R}\}, \\ E'_S &= \{(x, \beta, x\beta) \mid x, x\beta \in V'_S, \beta \in \Sigma^+, \\ &\qquad 1 \le \forall i < |\beta|, x \cdot \beta[1..i] \notin V'_S\}. \end{aligned}$$

Notice that the size of $STree'(y)$ is $O(n)$, since $|\mathcal{L} \cup \mathcal{R}| \le |V_S| + |V_D| = O(n)$, where V_S and V_D are respectively the sets of nodes of $STree(y)$ and $DAWG(y)$.

A node $x \in V'_S$ of $STree'(y)$ is called *black* iff $x \in \mathcal{R}$. See Figure 2 for an example of $STree'(y)$.

▶ **Lemma 8.** *For any $x \in Substr(y)$, if x is represented by a black node in $STree'(y)$, then every prefix of x is also represented by a black node in $STree'(y)$.*

Proof. Since x is a black node, $x = \overleftarrow{x}$. Assume on the contrary that there is a proper prefix z of x such that z is not represented by a black node. Let $zu = x$ with $u \in \Sigma^+$. Since $z \equiv_R \overleftarrow{z}$, we have $x = zu \equiv_R \overleftarrow{z}u$. On the other hand, since z is not black, we have $|\overleftarrow{z}| > |z|$. However, this contradicts that x is the longest member $\overleftarrow{x}$ of $[x]_R$. Thus, every prefix of x is also represented by a black node. ◀

▶ **Lemma 9.** *For any string y, let $BT(y)$ be the trie consisting only of the black nodes of $STree'(y)$. Then, every leaf ℓ of $BT(y)$ is a node of the original suffix tree $STree(y)$.*

Proof. Assume on the contrary that some leaf ℓ of $BT(y)$ corresponds to an internal node of $STree'(y)$ that has exactly one child. Since any substring in $\mathcal{L}$ is represented by a node of the original suffix tree $STree(y)$, we have $\ell \in \mathcal{R}$. Since $\ell = \overleftarrow{\ell}$, ℓ is the longest substring of y which has ending positions $EndPos(\ell)$ in y. This implies one of the following situations: (1) occurrences of ℓ in y are immediately preceded by at least two distinct characters $a \neq b$, (2) ℓ occurs as a prefix of y and all the other occurrences of ℓ in y are immediately preceded by a unique character a, or (3) ℓ occurs exactly once in y as its prefix. Let u be the only

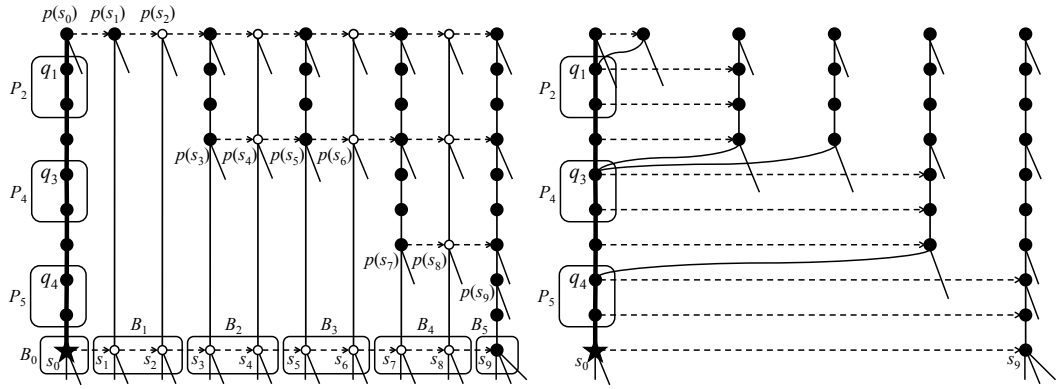

Figure 3 (Left): Illustration for a part of $STree'(y)$, where the branching nodes are those that exist also in the original suffix tree $STree(y)$. Suppose we have just visited node $x = s_0$ (marked by a star) in the post-order traversal on $STree'(y)$. Here, $s_0, \ldots, s_9$ are connected by a chain of the suffix links starting from s_0, and s_9 is the first black node after s_0 in the chain. In the corresponding DAG D, we will add in-coming edges to the black nodes in the path from $p(x)$ to x, and will add suffix links from these black nodes in the path. The sequence $s_0, \ldots, s_m$ of nodes in $STree'(y)$ is partitioned into blocks, such that that the parents of the nodes in the same block belong to the same equivalence class w.r.t. $\equiv_R$. (Right): The in-coming edges and the suffix links have been added to the nodes in the path from $p(x)$ to $x = s_0$.

child of ℓ in $STree'(y)$, and let $\ell z = u$, where $z \in \Sigma^+$. By the definition of ℓ, u is not black. On the other hand, in any of the situations (1)-(3), $u = \ell z$ is the longest substring of y which has ending positions $EndPos(u)$ in y. Hence we have $u = \overleftarrow{u}$ and u must be black, a contradiction. Thus, every leaf ℓ of $BT(y)$ is a node of the original suffix tree $STree(y)$. ◀

▶ **Lemma 10** ([14]). *For any node $x \in V_S$ of the original suffix tree $STree(y)$, its corresponding node in $STree'(y)$ is black iff (1) x is a leaf of the suffix link tree $SLT(y)$, or (2) x is an internal node of $SLT(y)$ and for any character $a \in \Sigma$ such that $ax \in V_S$, $|BegPos(ax)| \neq |BegPos(x)|$.*

Using Lemma 9 and Lemma 10, we can compute all leaves of $BT(y)$ in $O(n)$ time by a standard traversal on the suffix link tree $SLT(y)$. Then, we can compute all internal black nodes of $BT(y)$ in $O(n)$ time using Lemma 8. Now, by Theorem 7, the next lemma holds:

▶ **Lemma 11.** *Given a string y of length n over an integer alphabet, edge-sorted $STree'(y)$ can be constructed in $O(n)$ time.*

We construct $DAWG(y)$ with suffix links L_D from $STree'(y)$, as follows. First, we construct a DAG D, which is initially equivalent to the trie $BT(y)$ consisting only of the black nodes of $STree'(y)$. Our algorithm adds edges and suffix links to D, so that the DAG D will finally become $DAWG(y)$. In so doing, we traverse $STree'(y)$ in post-order. For each black node x of $STree'(y)$ visited in the post-order traversal, which is either an internal node or a leaf of the original suffix tree $STree(y)$, we perform the following: Let $p(x)$ be the parent of x in the *original suffix tree* $STree(y)$. It follows from Lemma 8 that every prefix x' of x with $|p(x)| \leq |x'| \leq |x|$ is represented by a black node. For each black node x' in the path from $p(x)$ to x in the DAG D, we compute the in-coming edges to x' and the suffix link of x'.

Let $s_0, \ldots, s_m$ be the sequence of nodes connected by a chain of suffix links starting from $s_0 = x$, such that $|BegPos(s_i)| = |BegPos(s_0)|$ for all $0 \leq i \leq m-1$ and $|BegPos(s_m)| > |BegPos(s_0)|$ (see the left diagram of Figure 3). In other words, s_m is the first black node

after s_0 in the chain of suffix links (this is true by Lemma 10). Since $|s_i| = |s_{i-1}|+1$ for every $1 \leq i \leq m-1$, $EndPos(s_i) = EndPos(s_0)$. Thus, $s_0, \ldots, s_{m-1}$ form a single equivalence class w.r.t. $\equiv_R$ and are represented by the same node as $x = s_0$ in the DAWG.

For any $0 \leq i \leq m-1$, let $d(s_i) = |s_i| - |p(s_i)|$. Observe that the sequence $d(s_0), \ldots, d(s_m)$ is monotonically non-increasing. We partition the sequence $s_0, \ldots, s_m$ of nodes into blocks so that the parents of all nodes in the same block belong to the same equivalence class w.r.t. $\equiv_R$. Let r be the number of such blocks, and for each $0 \leq k \leq r-1$, let $B_k = s_{i_k}, \ldots, s_{i_{k+1}-1}$ be the kth such block. Note that for each block B_k, $p(s_{i_k})$ is the only black node among the parents $p(s_{i_k}), \ldots, p(s_{i_{k+1}-1})$ of the nodes in B_k, since it is the longest one in its equivalence class $[p(s_{i_k})]_R$. Also, every node in the same block has the same value for function d. Thus, for each block B_k, we add a new edge $(p(s_{i_k}), b_k, q_k)$ to the DAG D, where q_k is the (black) ancestor of x such that $|q_k| = |x| - d(s_{i_k}) + 1$, and b_k is the first character of the label of the edge from $p(s_{i_k})$ to s_{i_k} in $STree'(y)$. Notice that this new edge added to D corresponds to the edges between the nodes in the block B_k and their parents in $STree'(y)$. We also add a suffix link $(p(q_k), a, p(s_{i_k}))$ to D, where $a = s_{i_k-1}[1]$. See also the right diagram of Figure 3.

For each $2 \leq k \leq r-1$, let P_k be the path from q_{k-1} to g_k, where $g_k = p(p(q_k))$ for $2 \leq k \leq r-2$ and $g_{r-1} = x = s_0$. Each P_k is a sub-path of the path from $p(s_0)$ to s_0, and every node in P_k has not been given their suffix link yet. For each node v in P_k, we add the suffix link from v to the ancestor u of s_{i_k} such that $|s_{i_k}| - |u| = |s_0| - |v|$. See also the right diagram of Figure 3.

Repeating the above procedure for all black nodes of $STree'(y)$ that are either internal nodes or leaves of the original suffix tree $STree(y)$ in post order, the DAG D finally becomes $DAWG(y)$ with suffix links L_D. We remark however that the edges of $DAWG(y)$ might not be sorted, since the edges that exist in $STree'(y)$ were firstly inserted to the DAG D. Still, we can easily sort all the edges of $DAWG(y)$ in $O(n)$ total time after they are constructed: First, extract all edges of $DAWG(y)$ by a standard traversal on $DAWG(y)$, which takes $O(n)$ time. Next, radix sort them by their labels, which takes $O(n)$ time because we assumed an integer alphabet of polynomial size in n. Finally, re-insert the edges to their respective nodes in the sorted order.

▶ **Theorem 12.** *Given a string y of length n over an integer alphabet, we can compute edge-sorted $DAWG(y)$ with suffix links L_D in $O(n)$ time and space.*

Proof. The correctness can easily be seen if one recalls that minimizing $STrie(y)$ based on its suffix links produces $DAWG(y)$. The proposed algorithm simulates this minimization using only the subset of the nodes of $STrie(y)$ that exist in $STree'(y)$. The out-edges of each node of $DAWG(y)$ are sorted in lexicographical order as previously described.

We analyze the time complexity of our algorithm. We can compute $STree'(y)$ in $O(n)$ time by Lemma 11. The initial trie for D can easily be computed in $O(n)$ time from $STree'(y)$. Let x be any node visited in the post-order traversal on $STree'(y)$ that is either an internal node or a leaf of the original suffix tree $STree(y)$. The cost of adding the new in-coming edges to the black nodes in the path from $p(x)$ to $x = s_0$ is linear in the number of nodes in the sequence $s_0, \ldots, s_m$ connected by the chain of suffix links starting from $s_0 = x$. Since s_0 and s_m are the only black nodes in the sequence, it follows from Lemma 10 that the chain of suffix links from s_0 to s_m is a non-branching path of the suffix link tree $SLT(y)$. This implies that the suffix links in this chain are used only for node x during the post-order traversal of $STree'(y)$. Since the number of edges in $SLT(y)$ is $O(n)$, the amortized cost of adding each edge to D is constant. Also, the total cost to sort all edges is $O(n)$, as was previously explained. Now let us consider the cost of adding the suffix links from the nodes in each sub-path P_k. For each node v in P_k, the destination node v can be found in constant time

by simply climbing up the path from s_{i_k} in the chain of suffix links. Overall, the total time cost to transform the trie for D to $DAWG(y)$ is $O(n)$.

The working space is clearly $O(n)$. ◀

Figure 4 shows an example of DAWG construction by our algorithm.

In some applications such as bidirectional pattern searches, it is preferable that the in-coming suffix links at each node of $DAWG(y)$ are also sorted in lexicographical order, but our algorithm described above does not sort the suffix links. However, we can sort the suffix links in $O(n)$ time by the same technique applied to the edges of $DAWG(y)$.

4 Constructing Affix Trees in $O(n)$ Time for Integer Alphabet

Let y be the input string of length n over an integer alphabet. Recall the sets $\mathcal{L} = \{\overrightarrow{x} \mid x \in Substr(y)\}$ and $\mathcal{R} = \{\overleftarrow{x} \mid x \in Substr(y)\}$ introduced in Section 3. For any set $S \subseteq \Sigma^* \times \Sigma^*$ of ordered pairs of strings, let $S[1] = \{x_1 \mid (x_1, x_2) \in S \text{ for some } x_2 \in \Sigma^*\}$ and $S[2] = \{x_2 \mid (x_1, x_2) \in S \text{ for some } x_1 \in \Sigma^*\}$. For any string x, let $\hat{x}$ denote the reversed string of x.

The affix tree [15] of string y, denoted $ATree(y)$, is a *bidirectional* text indexing structure defined as follows:

▶ **Definition 13.** $ATree(y)$ for string y is an edge-labeled DAG $(V_A, E_A) = (V_A, E_A^F \cup E_A^B)$ which has two mutually distinct sets E_A^F, E_A^B of edges such that

$$
\begin{aligned}
V_A &= \{(x, \hat{x}) \mid x \in \mathcal{L} \cup \mathcal{R}\},\\
E_A^F &= \{((x, \hat{x}), \beta, (x\beta, \hat{\beta}\hat{x})) \mid x, x\beta \in V_A[1],\\
&\qquad \beta \in \Sigma^+, 1 \leq \forall i < |\beta|, x \cdot \beta[1..i] \notin V_A[1]\},\\
E_A^B &= \{((x, \hat{x}), \hat{\alpha}, (\alpha x, \hat{x}\hat{\alpha})) \mid \hat{x}, \hat{x}\hat{\alpha} \in V_A[2],\\
&\qquad \alpha \in \Sigma^+, 1 \leq \forall i < |\alpha|, \hat{x} \cdot \hat{\alpha}[1..i] \notin V_A[2]\}.
\end{aligned}
$$

E_A^F is the set of forward edges labeled by substrings of y, while E_A^B is the set of backward edges labeled by substrings of $\hat{y}$.

▶ **Theorem 14.** *Given a string y of length n over an integer alphabet, we can compute edge-sorted $ATree(y)$ in $O(n)$ time and space.*

Proof. Clearly, there is a one-to-one correspondence between each node $(x, \hat{x}) \in V_A$ of $ATree(y) = (V_A, E_A^F \cup E_A^B)$ and each node $x \in V'_S$ of $STree'(y) = (V'_S, E'_S)$ of Section 3 (see also Figure 2 and Figure 5). Moreover, there is a one-to-one correspondence between each forward edge $(x, \beta, x\beta) \in E_A^F$ of $ATree(y)$ and each edge $(x, \beta, x\beta) \in E'_S$ of $STree'(y)$. Hence, what remains is to construct the backward edges in E_A^B for $ATree(y)$. A straightforward modification to our DAWG construction algorithm of Section 3 can construct the backward edges of $ATree(y)$; instead of working on the DAG D, we directly add the suffix links to the black nodes of $STree'(y)$ whose suffix links are not defined yet (namely, those that are neither branching nodes nor leaves of the suffix link tree $SLT(y)$). Since the suffix links are reversed edges, by reversing them we obtain the backward edges of $ATree(y)$. The labels of the backward edges can be easily computed in $O(n)$ time by storing in each node the length of the string it represents. Finally, we can sort the forward and backward edges in lexicographical order in overall $O(n)$ time, using the same idea as in Section 3. ◀

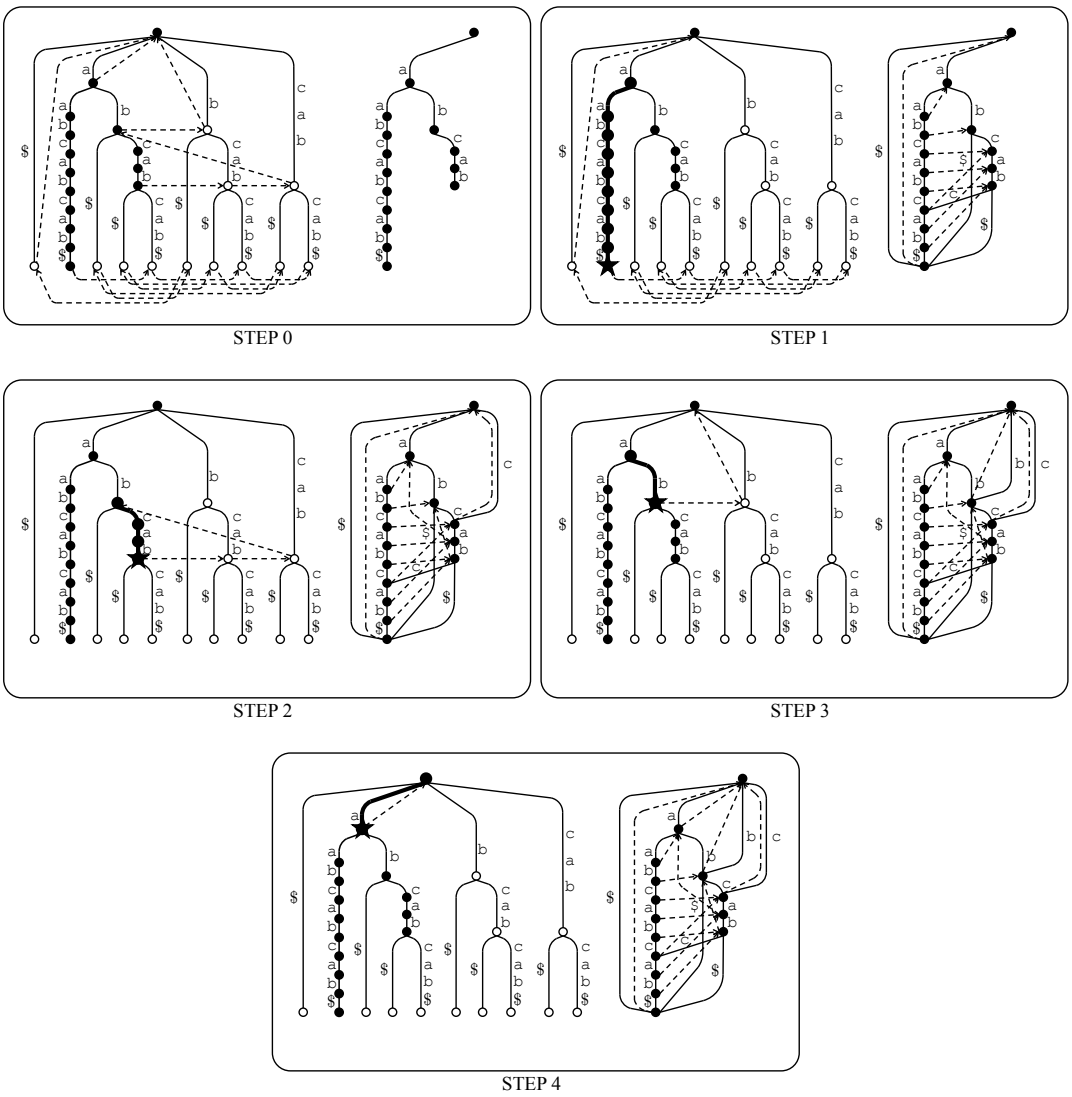

Figure 4 Snapshots during the construction of $DAWG(y)$ for $y = \texttt{aabcabcab\$}$. Step 0: (Left): $STree'(y)$ with suffix links L_S and (Right): the initial trie for D. We traverse $STree'(y)$ in post order. Step 1: We arrived at black leaf node $x_1 = \texttt{aabcabcab\$}$ (indicated by a star). We determine the in-coming edges and suffix links for the black nodes in the path from $p(x_1) = \texttt{a}$ and x_1 (indicated by thick black lines). To the right is the resulting DAG D for this step. Step 2: We arrived at black branching node $x_2 = \texttt{abcab}$ (indicated by a star). We determine the in-coming edges and suffix links for the black nodes in the path from $p(x_2) = \texttt{ab}$ and x_2 (indicated by thick black lines). To the right is the resulting DAG D for this step. Step 3: We arrived at black branching node $x_3 = \texttt{ab}$ (indicated by a star). We determine the in-coming edges and suffix links for the black nodes in the path from $p(x_3) = \texttt{a}$ and x_3 (indicated by thick black lines). To the right is the resulting DAG D for this step. Step 4: We arrived at black branching node $x_4 = \texttt{a}$ (indicated by a star). We determine the in-coming edges and suffix links for the black nodes in the path from $p(x_4) = \varepsilon$ and x_4 (indicated by thick black lines). To the right is the resulting DAG D for this step. Since all branching and leaf black nodes have been processed, the final DAG D is $DAWG(y)$ with suffix links.

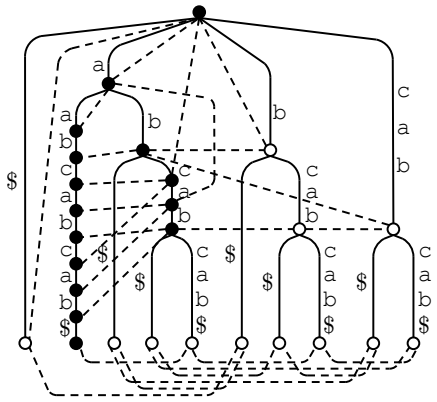

Figure 5 An example of $ATree(y)$ with string $y =$ aabcabcab\$. The solid arcs represent the forward edges in E_A^F, while the broken arcs represent the backward edges in E_A^B. For simplicity, the labels of backward edges are omitted.

5 Computing Minimal Absent Words in $O(n + |MAW(y)|)$ Time

As an application to our $O(n)$-time DAWG construction algorithm of Section 3, in this section we show an optimal time algorithm to compute the set of all minimal absent words of a given string over an integer alphabet.

Finding minimal absent words of length 1 for a given string y (i.e., the characters not occurring in y) is easy to do in $O(n+\sigma)$ time and $O(1)$ working space for an integer alphabet, where σ is the alphabet size. In what follows, we concentrate on finding minimal absent words of y of length at least 2.

Crochemore et al. [7] proposed a $\Theta(\sigma n)$-time algorithm to compute $MAW(y)$ for a given string y of length n. The following two lemmas, which show tight connections between $DAWG(y)$ and $MAW(y)$, are implicitly used in their algorithm but under a somewhat different formulation. Since our $O(n + |MAW(y)|)$-time solution is built on the lemmas, we give a proof for completeness.

▶ **Lemma 15.** *Let $a, b \in \Sigma$ and $x \in \Sigma^*$. If $axb \in MAW(y)$, then $x = \overleftarrow{x}$, namely, x is the longest string represented by node $[x]_R \in V_D$ of $DAWG(y)$.*

Proof. Assume on the contrary that $x \neq \overleftarrow{x}$. Since x is not the longest string of $[x]_R$, there exists a character $c \in \Sigma$ such that $cx \in Substr(y)$ and $[x]_R = [cx]_R$. Since $axb \in MAW(y)$, it follows from Lemma 6 that $xb \in Substr(y)$. Since $[x]_R = [cx]_R$, c always immediately precedes x in y. Thus we have $cxb \in Substr(y)$.

Since $axb \in MAW(y)$, $c \neq a$. On the other hand, it follows from Lemma 6 that $ax \in Substr(y)$. However, this contradicts that c always immediately precedes x in y and $c \neq a$. Consequently, if $axb \in MAW(y)$, then $x = \overleftarrow{x}$. ◀

For any node $v \in V_D$ of $DAWG(y)$ and character $b \in \Sigma$, we write $\delta_D(v, b) = u$ if $(v, b, u) \in E_D$ for some $u \in V_D$, and write $\delta_D(v, b) = nil$ otherwise. For any suffix link $(u, a, v) \in L_D$ of $DAWG(y)$, we write $sl_D(u) = v$. Since there is exactly one suffix link coming out from each node $u \in V_D$ of $DAWG(y)$, the character a is unique for each node u.

▶ **Lemma 16.** *Let $a, b \in \Sigma$ and $x \in \Sigma^*$. Then, $axb \in MAW(y)$ iff $x = \overleftarrow{x}$, $\delta_D([x]_R, b) = [xb]_R$, $sl_D([ax]_R) = [x]_R$, and $\delta_D([ax]_R, b) = nil$.*

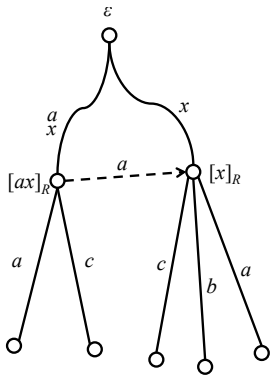

Figure 6 Computing minimal absent words from a DAWG. In this case, axb is a MAW since it does not occur in the string while ax and xb do.

Algorithm 1: $\Theta(n\sigma)$-time algorithm (MF-TRIE) by Crochemore et al. [7]

Input: String y of length n
Output: All minimal absent words for y

```
1 MAW ← ∅;
2 Construct DAWG(y) augmented with suffix links L_D;
3 for each non-source node u of DAWG(y) do
4     for each character b ∈ Σ do
5         if δ_D(u, b) = nil and δ_D(sl_D(u), b) ≠ nil then
6             MAW ← MAW ∪ {axb} ;        // (u, a, sl_D(u)) ∈ L_D , x = long(sl_D(u))
7 Output MAW;
```

Proof. ($\Rightarrow$) From Lemma 15, $x = \overleftarrow{x}$. From Lemma 6, $axb \notin Substr(y)$. However, $ax, xb \in Substr(y)$, and thus we have $\delta_D([ax]_R, b) = nil$, $\delta_D([x]_R, b) = [xb]_R$, and $sl_D([ax]_R) = [x]_R$, where the last suffix link exists since $x = \overleftarrow{x}$.

($\Leftarrow$) Since $\delta_D([x]_R, b) = [xb]_R$ and $sl_D([ax]_R) = [x]_R$, we have that $xb, ax \in Substr(y)$. Since $ax \in Substr(y)$ and $\delta_D([ax]_R, b) = nil$, we have that $axb \notin Substr(y)$ Thus from Lemma 6, $axb \in MAW(y)$. ◀

From Lemma 16 all MAWs of y can be computed by traversing all the states of $DAWG(y)$ and comparing all out-going edges between nodes connected by suffix links. A pseudo-code of the algorithm MF-TRIE by Crochemore et al. [7], which is based on this idea, is shown in Algorithm 1. Since all characters in the alphabet Σ are tested at each node, the total time complexity becomes $\Theta(n\sigma)$. The working space is $O(n)$, since only the DAWG and its suffix links are needed.

Next we show that with a simple modification in the for loops of the algorithm and with a careful examination of the total cost, the set $MAW(y)$ of all MAWs of the input string y can be computed in $O(n + |MAW(y)|)$ time and $O(n)$ working space. Basically, the only change is to move the "$\delta_D(sl_D(u), b) \neq nil$" condition in Line 5 to the for loop of Line 4. Namely, when we focus on node u of $DAWG(y)$, we test only the characters which label the out-edges from node $sl_D(u)$. A pseudo-code of the modified version is shown in Algorithm 2.

▶ **Theorem 17.** *Given a string y of length n over an integer alphabet, we compute $MAW(y)$ in optimal $O(n + |MAW(y)|)$ time with $O(n)$ working space.*

Algorithm 2: Proposed $O(n + |MAW(y)|)$-time algorithm

Input: String y of length n
Output: All minimal absent words for y

```
1 MAW ← ∅;
2 Construct edge-sorted DAWG(y) augmented with suffix links L_D;
3 for each non-source node u of DAWG(y) do
4     for each character b such that δ_D(sl_D(u), b) ≠ nil do
5         if δ_D(u, b) = nil then
6             MAW ← MAW ∪ {axb} ;        // (u, a, sl_D(u)) ∈ L_D, x = long(sl_D(u))
7 Output MAW;
```

Proof. First, we show the correctness of our algorithm. For any node u of $DAWG(y)$, $EndPos(sl_D(u)) \supset EndPos(u)$ holds since every string in $sl_D(u)$ is a suffix of the strings in u. Thus, if there is an out-edge of u labeled c, then there is an out-edge of $sl_D(u)$ labeled c. Hence, the task is to find every character b such that there is an out-edge of $sl_D(u)$ labeled b but there is no out-edge of u labeled b. The for loop of Line 4 of Algorithm 2 tests all such characters and only those. Hence, Algorithm 2 computes $MAW(y)$ correctly.

Second, we analyze the efficiency of our algorithm. As was mentioned above, minimal absent words of length 1 for y can be found in $O(n + \sigma)$ time and $O(1)$ working space. By Lemma 5, $\sigma \leq |MAW(y)|$ and hence the σ-term is dominated by the output size $|MAW(y)|$. Now we consider the cost of finding minimal absent words of length at least 2 by Algorithm 2. Let b be any character such that there is an out-edge e of $sl_D(u)$ labeled b. There are two cases: (1) If there is no out-edge of u labeled b, then we output an MAW, so we can charge the cost to check e to an output. (2) If there is an out-edge e' of u labeled b, then the trick is that we can charge the cost to check e to e'. Since each node u has exactly one suffix link going out from it, each out-edge of u is charged only once in Case (2). Since the out-edges of every node u and those of $sl_D(u)$ are both sorted, we can compute their difference for every node u in $DAWG(y)$, in overall $O(n)$ time. Edge-sorted $DAWG(y)$ with suffix links can be constructed in $O(n)$ time for an integer alphabet as in Section 3. Overall, Algorithm 2 runs in $O(n + |MAW(y)|)$ time. The space requirement is clearly $O(n)$. ◀

References

1 Golnaz Badkobeh, Maxime Crochemore, and Chalita Toopsuwan. Computing the maximal-exponent repeats of an overlap-free string in linear time. In *SPIRE 2012*, pages 61–72, 2012.

2 Djamal Belazzougui, Fabio Cunial, Juha Kärkkäinen, and Veli Mäkinen. Versatile succinct representations of the bidirectional burrows-wheeler transform. In *Proc. ESA 2013*, pages 133–144, 2013.

3 Anselm Blumer, J. Blumer, David Haussler, Andrzej Ehrenfeucht, M. T. Chen, and Joel I. Seiferas. The smallest automaton recognizing the subwords of a text. *Theor. Comput. Sci.*, 40:31–55, 1985. doi:10.1016/0304-3975(85)90157-4.

4 Anselm Blumer, J. Blumer, David Haussler, Ross M. McConnell, and Andrzej Ehrenfeucht. Complete inverted files for efficient text retrieval and analysis. *J. ACM*, 34(3):578–595, 1987. doi:10.1145/28869.28873.

5 Maxime Crochemore. Transducers and repetitions. *Theor. Comput. Sci.*, 45(1):63–86, 1986.

6 Maxime Crochemore, Gabriele Fici, Robert Mercas, and Solon P. Pissis. Linear-time sequence comparison using minimal absent words & applications. In *LATIN 2016*, pages 334–346, 2016.

7 Maxime Crochemore, Filippo Mignosi, and Antonio Restivo. Automata and forbidden words. *Inf. Process. Lett.*, 67(3):111–117, 1998. doi:10.1016/S0020-0190(98)00104-5.

8 Martin Farach-Colton, Paolo Ferragina, and S. Muthukrishnan. On the sorting-complexity of suffix tree construction. *J. ACM*, 47(6):987–1011, 2000.

9 Shunsuke Inenaga, Hiromasa Hoshino, Ayumi Shinohara, Masayuki Takeda, Setsuo Arikawa, Giancarlo Mauri, and Giulio Pavesi. On-line construction of compact directed acyclic word graphs. *Discrete Applied Mathematics*, 146(2):156–179, 2005.

10 Moritz G. Maaß. Linear bidirectional on-line construction of affix trees. *Algorithmica*, 37(1):43–74, 2003.

11 Udi Manber and Eugene W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, 1993.

12 Edward M. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 23(2):262–272, 1976. doi:10.1145/321941.321946.

13 Filippo Mignosi, Antonio Restivo, and Marinella Sciortino. Words and forbidden factors. *Theor. Comput. Sci.*, 273(1-2):99–117, 2002.

14 Kazuyuki Narisawa, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Efficient computation of substring equivalence classes with suffix arrays. In *CPM 2007*, pages 340–351, 2007.

15 Jens Stoye. Affix trees. Technical Report Report 2000-04, Universität Bielefeld, 2000. URL: https://www.techfak.uni-bielefeld.de/~stoye/dropbox/report00-04.pdf.

16 Shiho Sugimoto, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Finding absent words from grammar compressed strings. In the Festschrift for Bořivoj Melichar, 2012.

17 Yuka Tanimura, Yuta Fujishige, Tomohiro I, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. A faster algorithm for computing maximal α-gapped repeats in a string. In *SPIRE 2015*, pages 124–136, 2015.

18 Peter Weiner. Linear pattern matching algorithms. In *14th Annual Symposium on Switching and Automata Theory, Iowa City, Iowa, USA, October 15-17, 1973*, pages 1–11, 1973. doi:10.1109/SWAT.1973.13.

19 Jun-ichi Yamamoto, Tomohiro I, Hideo Bannai, Shunsuke Inenaga, and Masayuki Takeda. Faster compact on-line Lempel-Ziv factorization. In *STACS 2014*, pages 675–686, 2014.

20 J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, IT-23(3):337–343, 1977.