Sparse Compact Directed Acyclic Word Graphs

Shunsuke Inenaga[1,2] and Masayuki Takeda[2,3]

[1] Japan Society for the Promotion of Science
[2] Department of Informatics, Kyushu University, Japan
{shunsuke.inenaga, takeda}@i.kyushu-u.ac.jp
[3] SORST, Japan Science and Technology Agency (JST)

Abstract. The suffix tree of string w represents all suffixes of w, and thus it supports full indexing of w for exact pattern matching. On the other hand, a *sparse suffix tree* of w represents only a subset of the suffixes of w, and therefore it supports sparse indexing of w. There has been a wide range of applications of sparse suffix trees, e.g., natural language processing and biological sequence analysis. *Word suffix trees* are a variant of sparse suffix trees that are defined for strings that contain a special word delimiter #. Namely, the word suffix tree of string $w = w_1 w_2 \cdots w_k$, consisting of k words each ending with #, represents only the k suffixes of w of the form $w_i \cdots w_k$. Recently, we presented an algorithm which builds word suffix trees in $O(n)$ time with $O(k)$ space, where n is the length of w. In addition, we proposed *sparse directed acyclic word graphs* (*SDAWGs*) and an on-line algorithm for constructing them, working in $O(n)$ time and space. As a further achievement of this research direction, this paper introduces yet a new text indexing structure named *sparse compact directed acyclic word graphs* (*SCDAWGs*). We show that the size of SCDAWGs is smaller than that of word suffix trees and SDAWGs, and present an SCDAWG construction algorithm that works in $O(n)$ time with $O(k)$ space and in an on-line manner.

1 Introduction

Suffix trees have played a very central role in combinatorial pattern matching as they have wide applications such as data compression [10, 12, 6] and bioinformatics [11, 2, 5]. Suffix trees are fairly useful since they can be constructed in linear time and space in the input string length [14]. On the other hand, there have been great demands to deal with the common case that only certain suffixes of the input string are relevant. Suffix trees that contain only a subset of all suffixes are called *sparse suffix trees*. Among several versions of sparse suffix trees, we in this paper concentrate on *word suffix trees* introduced in [1].

Let D be a dictionary of words and w be a string in D^+ of length n, namely, w be a sequence $w_1 \cdots w_k$ of k words in D. The word suffix tree of w w.r.t. D is a tree structure which represents only the k suffixes of w in the form $w_i \cdots w_k$. Although the normal suffix tree of w requires $O(n)$ space, the word suffix tree of w w.r.t. D needs only $O(k)$ space. One typical application of word suffix trees is a word- and phrase-level index for documents written in a natural language. Note that normal suffix trees report *all* occurrences of a keyword in the text string, which may cause unwanted matchings (e.g., an occurrence of "other" in "mother" is possibly retrieved). Andersson et al. introduced an algorithm to build the word suffix tree for w w.r.t. D with $O(k)$ space, but in $O(n)$ *expected* time [1]. Lately, we invented a faster algorithm that constructs word suffix trees with $O(k)$ space and in $O(n)$ time *in the worst case* [8]. This is optimal, since the whole string w needs to be read at least once.

It is noteworthy that our word suffix tree construction algorithm gives a practical solution to linear-time construction of sparse suffix trees for arbitrary subsets of suffixes. Given a set S of $k-1$ positions in string w, we insert a unique word delimiter # into w at the positions listed in S. Now we get a string which consists of k words, each separated by #. The word suffix tree of this modified string is alternative to the sparse suffix tree of w w.r.t. S. In the matching phase, we simply ignore any #'s in the edge labels of the tree.

In this paper, we introduce a new data structure named *sparse compact directed acyclic word graphs* (*SCDAWGs*) as an alternative to the word suffix trees and to the sparse suffix trees as well. SCDAWGs are a sparse text indexing version of *compact directed acyclic word graphs* (*CDAWGs*) of [4]. We define SCDAWGs based on 'word-position-sensitive' equivalence relations on string w and dictionary D, and show the asymptotic size of SCDAWGs to be $O(k)$. Moreover, the fact is that SCDAWGs are a minimization of sparse suffix trees, and therefore require no more space than sparse suffix trees. Finally, we present an on-line algorithm for building SCDAWGs, which is based on the on-line algorithm for building normal CDAWGs in [7]. By using the minimum DFA M_D which accepts D, and by tailoring suffix links accordingly, the modified algorithm constructs SCDAWGs. Since our algorithm directly constructs SCDAWGs (namely, not constructing sparse suffix trees as an intermediate), it works with space linear in the output size. We also show that the proposed algorithm runs in $O(n)$ time. Furthermore, our algorithm can be seen as a generalization of the normal CDAWG construction algorithm of [7]. Assume just for now $D = \Sigma$, and consider a DFA which accepts Σ with only two states that are a single initial state and a single final state. Then this DFA plays the same role as the auxiliary '$\perp$' node used in [7], and as a result normal CDAWGs are generated.

2 Preliminaries

2.1 Notations

Let Σ be a finite set of symbols, called an *alphabet*. Throughout this paper we assume that Σ is fixed. A finite sequence of symbols is called a *string*. We denote the length of string w by $|w|$. The empty string is denoted by ε, that is, $|\varepsilon| = 0$. Let Σ^* be the set of strings over Σ. For any symbol $a \in \Sigma$, we define a^{-1} such that $a^{-1}a = \varepsilon$.

Strings x, y, and z are said to be a *prefix*, *substring*, and *suffix* of string $w = xyz$, respectively. A prefix, substring, and suffix of string w are said to be *proper* if they are not w. Let *Prefix*(w) and *Suffix*(w) be the set of the prefixes and suffixes of string w, respectively. For set S of strings, let $\mathit{Prefix}(S) = \bigcup_{w \in S} \mathit{Prefix}(w)$.

Definition 1 (Prefix property). *A set L of strings is said to satisfy the* prefix property *if no string in L is a proper prefix of another string in L.*

The i-th symbol of string w is denoted by $w[i]$ for $1 \le i \le |w|$, and the substring of string w that begins at position i and ends at position j is denoted by $w[i..j]$ for $1 \le i \le j \le |w|$. For convenience, let $w[i..j] = \varepsilon$ for $i > j$. For any strings $x, w \in \Sigma^*$, let

$$\begin{aligned} \mathit{Begpos}_w(x) &= \{i \mid x = w[i..i+|x|-1]\}, \text{ and} \\ \mathit{Endpos}_w(x) &= \{j \mid x = w[j-|x|+1..j]\}. \end{aligned}$$

Let D be a set of strings called a *dictionary*. A *factorization* of string w w.r.t. D is a list $w_1, \ldots, w_k$ of strings in D such that $w = w_1 \cdots w_k$ and $w_i \in D$ for each

$1 \leq i \leq k$. In the rest of the paper, we assume that $D = \Sigma^* \#$ where # is a special symbol not belonging to Σ, and that $w \in D^+$. Then, a factorization of w w.r.t. D is always unique, since D clearly satisfies the prefix property because of # not being in Σ.

For any string $w = w_1 \cdots w_k \in D^+$, let u be any prefix of w. Then we can write as $u = w_1 \cdots w_\ell v$ with $1 \leq \ell < k$, where v is a prefix of $w_{\ell+1}$. For any $1 \leq i \leq \ell$, let $u_i = w_i \cdots w_\ell v$, and for convenience, let $u_{\ell+1} = v$ and $u_{\ell+2} = \varepsilon$. Now, we define a word-oriented subset $\mathit{Suffix}_D(u)$ of $\mathit{Suffix}(u)$ as follows:

$$\mathit{Suffix}_D(u) = \{u_i \mid 1 \leq i \leq \ell + 2\}.$$

Namely, $\mathit{Suffix}_D(u)$ consists only of u, the suffixes of u which immediately follow any # in u, and the empty string ε.

Example 2. Let $\Sigma = \{\mathtt{a}, \mathtt{b}\}$, $D = \Sigma^*\#$, $w = \mathtt{ab\#b\#aa\#}$, and $u = \mathtt{ab\#b\#a}$. Then $\mathit{Suffix}_D(u) = \{\mathtt{ab\#b\#a}, \mathtt{b\#a}, \mathtt{a}, \varepsilon\}$.

Further, we define set $\mathit{Wordpos}_D(u)$ of the word-starting positions in u, as follows:

$$\mathit{Wordpos}_D(u) = \{|u| - |s| + 1 \mid s \in \mathit{Suffix}_D(u) - \{\varepsilon\}\}.$$

Example 3. For the running string $u = \mathtt{ab\#b\#a}$, $\mathit{Wordpos}_D(u) = \{1, 4, 6\}$.

2.2 Equivalence Classes on Strings over *D*

For set S of integers and integer i, we denote $S \oplus i = \{j + i \mid j \in S\}$ and $S \ominus i = \{j - i \mid j \in S\}$. Let $w \in D^+$. For any $u \in \mathit{Prefix}(\mathit{Suffix}_D(w))$ and $x, y \in (\Sigma \cup \{\#\})^*$, we define the begin- and end-equivalence relations $\equiv_u^B$ and $\equiv_u^E$, as follow:

$$\begin{aligned} x \equiv_u^B y &\Leftrightarrow \mathit{Begpos}_u(x) \cap \mathit{Wordpos}_D(u) = \mathit{Begpos}_u(y) \cap \mathit{Wordpos}_D(u), \\ x \equiv_u^E y &\Leftrightarrow \mathit{Endpos}_u(x) \cap (\mathit{Wordpos}_D(u) \oplus |x| \ominus 1) \\ &\qquad\qquad = \mathit{Endpos}_u(y) \cap (\mathit{Wordpos}_D(u) \oplus |y| \ominus 1). \end{aligned}$$

We note that the above equivalence relations are 'word-position-sensitive' versions of the equivalence relations introduced in [3], where the intersections with $\mathit{Wordpos}_D(u)$ make them word-position-sensitive. We denote by $[x]_u^B$ and $[x]_u^E$ the equivalence classes of x w.r.t. $\equiv_u^B$ and $\equiv_u^E$, respectively.

Proposition 4. *All strings that are not in $\mathit{Prefix}(\mathit{Suffix}_D(u))$ form one equivalence class under $\equiv_u^B$ (and $\equiv_u^E$), called the* degenerate class.

Proof. For any $x \notin \mathit{Prefix}(\mathit{Suffix}_D(u))$, clearly $\mathit{Wordpos}_D(u) = \emptyset$. Thus $\mathit{Begpos}_u(x) \cap \mathit{Wordpos}_D(u) = \emptyset$. Hence, any strings not belonging to $\mathit{Prefix}(\mathit{Suffix}_D(u))$ form one equivalence class. Similar discussion holds for $\equiv_u^E$. □

Proposition 5. *For any strings $x, y \in \mathit{Prefix}(\mathit{Suffix}_D(u))$, if $x \equiv_u^B y$, then either $x \in \mathit{Prefix}(y)$, or vice versa.*

Proof. Assume, without loss of generality, that $|x| \leq |y|$. Since $x \equiv_u^B y$, $\mathit{Begpos}_u(x) \cap \mathit{Wordpos}_D(u) = \mathit{Begpos}_u(y) \cap \mathit{Wordpos}_D(u)$. Let S be this set of positions. For any $i \in S$, we have that $x = u[i..i + |x| - 1]$ and $y = u[i..i + |y| - 1]$. Since $|x| \leq |y|$, $x \in \mathit{Prefix}(y)$. □

Example 6. For the running string $u = \mathtt{ab\#b\#a}$, $[\mathtt{b\#a}]^B_u = \{\mathtt{b\#a}, \mathtt{b\#}, \mathtt{b}\}$. For any pair of strings $x, y \in [\mathtt{b\#a}]^B_u$, we have $x \in \mathit{Prefix}(y)$ or $y \in \mathit{Prefix}(x)$.

Proposition 7. *For any strings $x, y \in \mathit{Prefix}(\mathit{Suffix}_D(u))$, if $x \equiv^E_u y$, then either $x \in \mathit{Suffix}_D(y)$, or vice versa.*

Proof. Assume, without loss of generality, that $|x| \le |y|$. Since $x \equiv^E_u y$, $\mathit{Endpos}_u(x) \cap (\mathit{Wordpos}_D(u) \oplus |x| \ominus 1) = \mathit{Endpos}_u(y) \cap (\mathit{Wordpos}_D(u) \oplus |y| \ominus 1)$. Let S be this set of positions. For any $i \in S$, we have that $x = u[i-|x|+1..i]$ and $y = u[i-|y|+1..i]$. Since $x \in \mathit{Prefix}(u[i-|x|+1..|u|])$, $u[i-|x|+1..|w|] \in \mathit{Suffix}_D(u)$, and $\mathit{Wordpos}_D(u) = \{|u| - |s| + 1 \mid s \in \mathit{Suffix}_D(u) - \{\varepsilon\}\}$, we have $i - |x| + 1 \in \mathit{Wordpos}_D(u)$. Similarly, $i - |y| + 1 \in \mathit{Wordpos}_D(u)$. Since $|x| \le |y|$, we have $x \in \mathit{Suffix}_D(y)$. □

Example 8. For the running string $u = \mathtt{ab\#b\#a}$, $[\mathtt{ab\#b}]^E_u = \{\mathtt{ab\#b}, \mathtt{b}\}$. Then $\mathtt{b} \in \mathit{Suffix}_D(\mathtt{ab\#b})$.

From Propositions 5 and 7, each non-degenerate equivalence class under $\equiv^B_u$ or $\equiv^E_u$ has a unique longest member, which is called the *representative* of the equivalence class. For any $x \in \mathit{Prefix}(\mathit{Suffix}_D(u))$, the representatives of $[x]^E_u$ and $[x]^B_u$ are denoted by $\overset{u}{\overleftarrow{x}}$ and $\overset{u}{\overrightarrow{x}}$, respectively.

For any $x \in \mathit{Prefix}(\mathit{Suffix}_D(u))$ such that $\overset{u}{\overrightarrow{x}} = x\alpha$ and $\overset{u}{\overleftarrow{x}} = \beta x$ with $\alpha, \beta \in (\Sigma \cup \{\#\})^*$, we denote $\overset{u}{\overleftrightarrow{x}} = \beta x \alpha$.

Proposition 9. *For any $x \in \mathit{Prefix}(\mathit{Suffix}_D(u))$, $\overset{u}{\overleftrightarrow{x}} = \overset{u}{\overleftarrow{(\overset{u}{\overrightarrow{x}})}} = \overset{u}{\overrightarrow{(\overset{u}{\overleftarrow{x}})}}$.*

Proof. Let $\overset{u}{\overrightarrow{x}} = x\alpha$ and $\overset{u}{\overleftarrow{x}} = \beta x$ with $\alpha, \beta \in (\Sigma \cup \{\#\})^*$. Then, $\overset{u}{\overleftrightarrow{x}} = \beta x \alpha$. Since $\overset{u}{\overrightarrow{x}} = x\alpha$, we have

$$
\begin{aligned}
x \equiv^B_u x\alpha &\Leftrightarrow \mathit{Begpos}_u(x) \cap \mathit{Wordpos}_D(u) = \mathit{Begpos}_u(x\alpha) \cap \mathit{Wordpos}_D(u) \\
&\Leftrightarrow (\mathit{Begpos}_u(x) \oplus |x\alpha| \ominus 1) \cap (\mathit{Wordpos}_D(u) \oplus |x\alpha| \ominus 1) \\
&\qquad = (\mathit{Begpos}_u(x\alpha) \oplus |x\alpha| \ominus 1) \cap (\mathit{Wordpos}_D(u) \oplus |x\alpha| \ominus 1) \\
&\Leftrightarrow (\mathit{Endpos}_u(x) \oplus |\alpha|) \cap (\mathit{Wordpos}_D(u) \oplus |x\alpha| \ominus 1) \\
&\qquad = \mathit{Endpos}_u(x\alpha) \cap (\mathit{Wordpos}_D(u) \oplus |x\alpha| \ominus 1).
\end{aligned} \tag{1}
$$

On the other hand, since $\overset{u}{\overleftarrow{x}} = \beta x$, we have

$$
\begin{aligned}
x \equiv^E_u \beta x &\Leftrightarrow \mathit{Endpos}_u(x) \cap (\mathit{Wordpos}_D(u) \oplus |x| \ominus 1) \\
&\qquad = \mathit{Endpos}_u(\beta x) \cap (\mathit{Wordpos}_D(u) \oplus |\beta x| \ominus 1) \\
&\Leftrightarrow (\mathit{Endpos}_u(x) \oplus |\alpha|) \cap (\mathit{Wordpos}_D(u) \oplus |x| \oplus |\alpha| \ominus 1) \\
&\qquad = (\mathit{Endpos}_u(\beta x) \oplus |\alpha|) \cap (\mathit{Wordpos}_D(u) \oplus |\beta x| \oplus |\alpha| \ominus 1) \\
&\Leftrightarrow (\mathit{Endpos}_u(x) \oplus |\alpha|) \cap (\mathit{Wordpos}_D(u) \oplus |x\alpha| \ominus 1) \\
&\qquad = (\mathit{Endpos}_u(\beta x) \oplus |\alpha|) \cap (\mathit{Wordpos}_D(u) \oplus |\beta x\alpha| \ominus 1).
\end{aligned} \tag{2}
$$

Let

$$
\begin{aligned}
A &= (\mathit{Endpos}_u(\beta x\alpha)) \cap (\mathit{Wordpos}_D(u) \oplus |\beta x\alpha| \ominus 1), \text{ and} \\
B &= (\mathit{Endpos}_u(\beta x) \oplus |\alpha|) \cap (\mathit{Wordpos}_D(u) \oplus |\beta x\alpha| \ominus 1)
\end{aligned}
$$

We show $A = B$. Since $\mathit{Endpos}_u(\beta x\alpha) \subseteq (\mathit{Endpos}_u(\beta x) \oplus |\alpha|)$, it is clear that $A \subseteq B$. For any $i \in B$, let $k = i - |x| - |\alpha| + 1$. Then $u[k..k + |x| - 1] = x$ and thus $k \in \mathit{Begpos}_u(x)$. Let $j = i - |\beta x| - |\alpha| + 1$. Then $u[j..j + |\beta x| - 1] = \beta x$. We have $j \in \mathit{Wordpos}_D(u)$ since $i \in B$. By Proposition 7 we have $x \in \mathit{Suffix}_D(\beta x)$, and therefore $\beta x[|\beta|] = \beta[|\beta|] = u[j + |\beta| - 1] = \#$. Hence $j + |\beta| \in \mathit{Wordpos}_D(u)$. On the other hand, $j + |\beta| = i - |\beta x| - |\alpha| + 1 + |\beta| = k$, thus $k \in \mathit{Wordpos}_D(u)$. Since $u[k..k + |x| - 1] = x$ and $\overset{u}{\overrightarrow{x}} = x\alpha$, $u[k..k + |x\alpha| - 1] = x\alpha$. Now we get

$$\begin{aligned} u[j..j + |\beta x\alpha| - 1] &= u[j..j + |\beta| - 1]u[j + |\beta|..j + |\beta x\alpha| - 1] \\ &= u[j..j + |\beta| - 1]u[k..k + |x\alpha| - 1] \\ &= \beta x\alpha. \end{aligned}$$

Thus $i = j + |\beta x\alpha| - 1 \in A$, and we obtain $B \subseteq A$.

From Equations (1) and (2), and $A = B$, we get $x\alpha \equiv^E_u \beta x\alpha$. It is easy to see that $\beta x\alpha$ is the representative of $[x\alpha]^E_u$. Finally, $\overset{u}{\overleftarrow{(\overset{u}{\overrightarrow{x}})}} = \overset{u}{\overleftarrow{x\alpha}} = \beta x\alpha = \overset{u}{\overleftrightarrow{x}}$. Similarly we can show $\overset{u}{\overrightarrow{(\overset{u}{\overleftarrow{x}})}} = \overset{u}{\overleftrightarrow{x}}$. □

3 Sparse Compact Directed Acyclic Word Graphs

In this section we introduce our new text indexing structure, *sparse compact directed acyclic word graphs* (*SCDAWGs*).

3.1 Definitions and Size Bounds

We first give a formal definition of sparse (word) suffix trees.

Definition 10 (Sparse suffix tree). *The* sparse suffix tree *of string $w \in D^+$, denoted by $\mathit{SSTree}_D(w)$, is a tree (V, E) such that*

$$\begin{aligned} V &= \{\overset{w}{\overrightarrow{x}} \mid x \in \mathit{Prefix}(\mathit{Suffix}_D(w))\}, \\ E &= \left\{ (\overset{w}{\overrightarrow{x}}, a\beta, \overset{w}{\overrightarrow{xa}}) \;\middle|\; \begin{array}{l} x, xa \in \mathit{Prefix}(\mathit{Suffix}_D(w)),\ a \in \Sigma \cup \{\#\}, \\ \beta \in (\Sigma \cup \{\#\})^*, \text{ and } \overset{w}{\overrightarrow{x}}a\beta = \overset{w}{\overrightarrow{xa}} \end{array} \right\}. \end{aligned}$$

Theorem 11 ([1]). *For any string $w = w_1 \cdots w_k \in D^+$, $\mathit{SSTree}_D(w)$ has $O(k)$ nodes and edges.*

To prove the above theorem, it suffices to show the two following claims:

Claim. $\mathit{SSTree}_D(w)$ has at most k leaves.

Proof. Recall that $w = w_1 \cdots w_k$ and that $|\mathit{Suffix}_D(x) - \{\varepsilon\}| = k$. For any $x \in \mathit{Prefix}(\mathit{Suffix}_D(w)) - (\mathit{Suffix}_D(w) - \{\varepsilon\})$, it follows from Definition 10 that there exist a symbol $a \in \Sigma \cup \{\#\}$ and a string $\beta \in (\Sigma \cup \{\#\})^*$ satisfying $\overset{w}{\overrightarrow{x}}a\beta = \overset{w}{\overrightarrow{xa}} \in \mathit{Prefix}(\mathit{Suffix}_D(w))$. Thus there can be at most k leaves in $\mathit{SSTree}_D(w)$. □

Claim. All internal nodes of $\mathit{SSTree}_D(w)$ are branching (of out-degree at least 2).

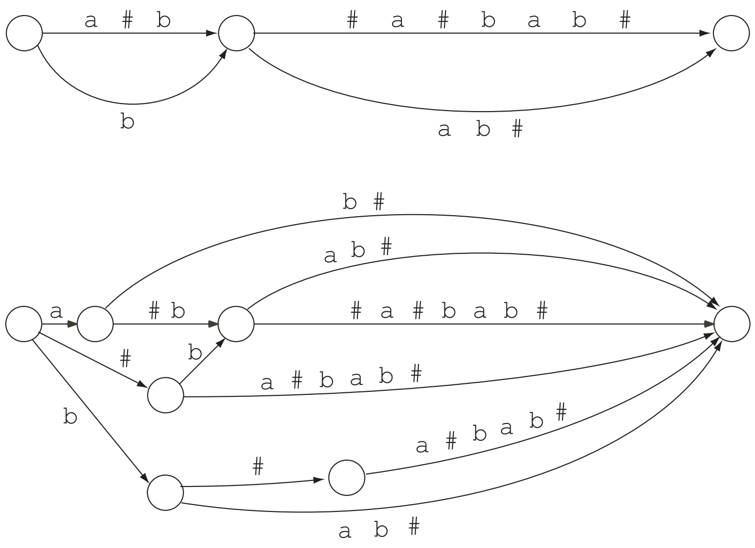

Figure 1. $SCDAWG_D(w)$ with $w = \mathtt{a\#b\#a\#bab\#}$ and $D = \{\mathtt{a}, \mathtt{b}\}^*\#$ is shown on the upper, and normal $CDAWG(w)$ is shown on the lower for comparison. Observe that $SCDAWG_D(w)$ contains only suffixes in $\mathit{Suffix}_D(w)$, while $CDAWG(w)$ has all the suffixes in $\mathit{Suffix}(w)$.

Proof. Assume for contrary that $\overset{w}{\overrightarrow{x}} = x$ and internal node $\overset{w}{\overrightarrow{x}}$ is not branching, i.e., there exists a unique symbol $a \in \Sigma \cup \{\#\}$ such that $\mathit{Begpos}_w(\overset{w}{\overrightarrow{x}}) = \mathit{Begpos}_w(\overset{w}{\overrightarrow{x}} \cdot a)$. Then we have $x \equiv^B_w xa$, which contradicts with the precondition that $\overset{w}{\overrightarrow{x}} = x$. Thus all internal nodes of $\mathit{SSTree}_D(w)$ are branching. □

Now we define *sparse compact directed acyclic word graphs* (*SCDAWGs*), as follows.

Definition 12 (Sparse compact directed acyclic word graph). *The* sparse compact directed acyclic word graph *of string $w \in D^+$, denoted by $SCDAWG_D(w)$, is a DAG (V, E) such that*

$$V = \{[\overset{w}{\overrightarrow{x}}]^E_w \mid x \in \mathit{Prefix}(\mathit{Suffix}_D(w))\},$$
$$E = \left\{ ([\overset{w}{\overrightarrow{x}}]^E_w, a\beta, [\overset{w}{\overrightarrow{xa}}]^E_w) \;\middle|\; \begin{array}{l} x, xa \in \mathit{Prefix}(\mathit{Suffix}_D(w)),\ a \in \Sigma \cup \{\#\}, \\ \beta \in (\Sigma \cup \{\#\})^*,\ \mathit{and}\ \overset{w}{\overrightarrow{x}} a\beta = \overset{w}{\overrightarrow{xa}} \end{array} \right\}.$$

$SCDAWG_D(w)$ has single *source node* $[\overset{w}{\overrightarrow{\varepsilon}}]^E_w = [\varepsilon]^E_w$ of in-degree zero, and single *sink node* $[\overset{w}{\overrightarrow{w}}]^E_w = [w]^E_w$ of out-degree zero.

We associate each node $[\overset{w}{\overrightarrow{x}}]^E_w$ of $SCDAWG_D(w)$ with $\mathit{length}([\overset{w}{\overrightarrow{x}}]^E_w) = |\overset{w}{\overleftarrow{(\overset{w}{\overrightarrow{x}})}}| = |\overset{w}{\overleftrightarrow{x}}|$.

Figure 1 shows $SCDAWG_D(w)$ with $w = \mathtt{a\#b\#a\#bab\#}$ and $D = \{\mathtt{a}, \mathtt{b}\}^*\#$, together with normal $CDAWG(w)$ representing all suffixes of w.

Due to the reflexivity of equivalence relations, for any string x, we have $x \in [x]^E_w$. Consequently, from Definitions 10 and 12, and Theorem 11, we obtain the following theorem regarding the asymptotic size bound of SCDAWGs.

Theorem 13. *For any string $w = w_1 \cdots w_k \in D^+$, $SCDAWG_D(w)$ has $O(k)$ nodes and edges.*

Notice that $SCDAWG_D(w)$ is a minimized version of $SSTree_D(w)$ by the end-equivalence classes. As a matter of fact, $SCDAWG_D(w)$ can be constructed by applying to $SSTree_D(w)$ the DAG minimization algorithm of [13], in time proportional to the number of edges in $SSTree_D(w)$. Due to [8], $SSTree_D(w)$ can be constructed in $O(n)$ time and $O(k)$ space, thus it is possible to build $SCDAWG_D(w)$ in $O(n)$ time and $O(k)$ space. However, this indirect construction wastes extra time and space of once building $SSTree_D(w)$. In the following section, we present our on-line, linear-time algorithm that directly constructs $SCDAWG_D(w)$ in $O(n)$ time and $O(k)$ space. Hence our algorithm consumes only linear space in the output size.

4 On-line Construction Algorithm for SCDAWGs

In this section we present our SCDAWG construction algorithm. Our algorithm is on-line, namely, it sequentially processes the input string $w \in D^+$ from left to right, one by one. To discuss this on-line construction, we extend the definition of $SCDAWG_D(\cdot)$ to any prefix u of $w \in D^+$, by replacing string w with its arbitrary prefix u in Definition 12.

4.1 Suffix Links

In this section we define the *suffix links* of SCDAWGs. We modify the suffix links of normal CDAWGs so that they are suitable for constructing SCDAWGs. The tailored suffix links are essential to the linearity of our SCDAWG construction algorithm.

Let us consider the minimum DFA M_D which accepts $D = \Sigma^*\#$. Then it is easy to see that M_D requires only constant space, with a unique final state (refer to the left of Figure 2). Let q_s and q_f be the initial and final states of M_D, respectively. Then we attach M_D to the SCDAWG, by replacing q_f with the source node of the SCDAWG. Now we are ready to define the suffix links of SCDAWGs.

Definition 14 (Suffix links of SCDAWGs). *For any node $[\overset{u}{\overrightarrow{x}}]^E_u$ of $SDAWG_D(u)$, let z be the shortest member of $[\overset{u}{\overrightarrow{x}}]^E_u$.*

1. *If $\overset{u}{\overrightarrow{x}} \neq u$ and $z \in \Sigma^*$, the suffix link from node $[\overset{u}{\overrightarrow{x}}]^E_u$ goes to the initial state q_s of M_D;*
2. *If $\overset{u}{\overrightarrow{x}} \neq u$ and $z \in (\Sigma \cup \{\#\})^+$, the suffix link from node $[\overset{u}{\overrightarrow{x}}]^E_u$ goes to to node $[y]^E_u$, where y is the longest string in $Suffix_D(z)$ such that $y \notin [\overset{u}{\overrightarrow{x}}]^E_u$;*
3. *Otherwise (If $\overset{u}{\overrightarrow{x}} = u$), the suffix link from $[\overset{u}{\overrightarrow{x}}]^E_u$ is undefined.*

The suffix link of the node in Group 3 is undefined, as it is never used in our construction algorithm to be shown later. See Figure 2 showing $SCDAWG_D(u)$ and its suffix links, where $u = \texttt{a\#b\#a\#bab\#b}$.

4.2 Updating $SCDAWG_D(u)$ to $SCDAWG_D(ua)$

In what follows, we consider to update $SCDAWG_D(u)$ to $SCDAWG_D(ua)$ with $u, ua \in Prefix(w)$, $a \in \Sigma \cup \{\#\}$, and $w \in D^+$.

The next proposition describes what happens to *Wordpos* and *Endpos*.

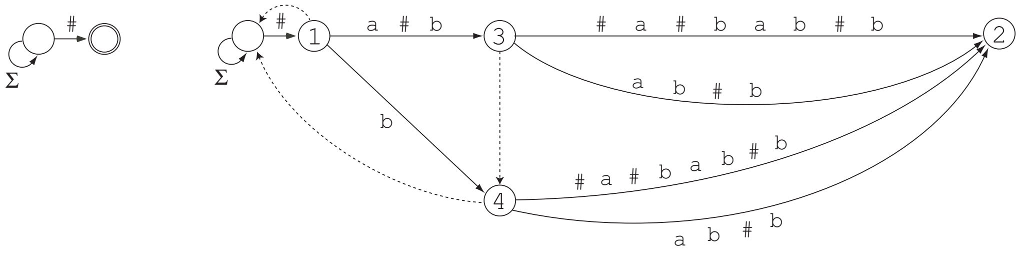

Figure 2. To the left is the minimum DFA M_D accepting dictionary $D = \Sigma^*\#$. To the right is $SCDAWG_D(u)$ with $u = \texttt{a\#b\#a\#bab\#b}$, where Node 1 is its source node. The suffix links are displayed by broken arrows. Nodes 1 and 4 are those of Group 1, Nodes 3 is that of Group 2, and Node 2 is that of Group 3 of Definition 14.

Proposition 15 ([9]). *Let $w \in D^+$ and $u, ua \in Prefix(w)$ with $a \in \Sigma \cup \{\#\}$. Then,*

$$Wordpos_D(ua) = \begin{cases} Wordpos_D(u) \cup \{|ua|\}, & \text{if } u[|u|] = \#; \\ Wordpos_D(u), & \text{otherwise.} \end{cases}$$

Also, for any string $x \in (\Sigma \cup \{\#\})^*$,

$$Endpos_{ua}(x) = \begin{cases} Endpos_u(x) \cup \{|ua|\}, & \text{if } x \in Suffix(ua); \\ Endpos_u(x), & \text{otherwise.} \end{cases}$$

From here on we consider what happens to the nodes of $SCDAWG_D(u)$ when updated to $SCDAWG_D(ua)$.

Proposition 16. *Let $w \in D^+$. For any $u, ua \in Prefix(w)$ with $a \in \Sigma \cup \{\#\}$, $\equiv^B_{ua}$ and $\equiv^E_{ua}$ are a refinement of $\equiv^B_u$ and $\equiv^E_u$, respectively.*

Let $lrs_D(ua)$ denote the longest string in $Suffix_D(ua) \cap Prefix(Suffix_D(u))$. Then we have:

Proposition 17. *Let $w \in D^+$. For any $u, ua \in Prefix(w)$ with $a \in \Sigma \cup \{\#\}$, $[ua]^E_{ua} = Suffix_D(u) \cdot a - Suffix_D(lrs_D(ua))$.*

The above proposition implies that the new sink node $[ua]^E_{ua}$ can be created by extending the incoming edges of the old sink node $[u]^E_u$ with symbol a, and by inserting a new a-labeled edge from each node $[s]^E_u$ to the old sink node, where $s \in Suffix_D(u) - Suffix_D(lrs_D(ua) \cdot a^{-1}) - [u]^E_u$.

Locating $lrs_D(ua)$ in the SCDAWG can be done according to the following proposition.

Proposition 18. *Let $w \in D^+$. For any u, $ua \in Prefix(w)$ with $a \in \Sigma \cup \{\#\}$, $lrs_D(ua) \in Suffix_D(lrs_D(u)) \cdot a$.*

The above proposition implies that we can locate $lrs_D(ua)$ by checking, for each $t \in Suffix_D(lrs_D(u))$, the transitivity from $[t]^E_u$ with new symbol a in the SCDAWG, in the decreasing order of the lengths. When we encounter the first string $y \in Suffix_D(lrs_D(u))$ such that $ya \in Suffix_D(ua)$, then ya is $lrs_D(ua)$. Locating $[t]^E_u$ can be done efficiently by using suffix links of Definition 14.

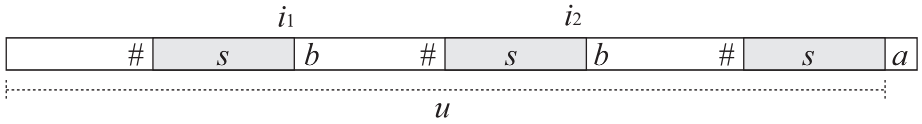

Figure 3. We postpone creating a node corresponding to $s \in \mathit{Suffix}_D(u) - \mathit{Suffix}_D(\mathit{lrs}_D(ua) \cdot a^{-1}) - [u]^E_u$ until we get the first symbol a satisfying $a \neq b = u[i{+}1]$ for any $i \in \mathit{Endpos}_u(s) \cap (\mathit{Wordpos}_D(u) \oplus |s| \ominus 1)$.

Creating new nodes. While searching for $\mathit{lrs}_D(ua)$, due to Proposition 17, we create a new a-labeled edge from the nodes corresponding to strings $s \in \mathit{Suffix}_D(u) - \mathit{Suffix}_D(\mathit{lrs}_D(ua) \cdot a^{-1}) - [u]^E_u$. However, the following proposition suggests a difficulty of linear time maintenance of those nodes at each stage of updating the SCDAWG.

Proposition 19. *Let $w \in D^+$ and $u \in \mathit{Prefix}(w)$. For any $t \in \mathit{Suffix}_D(u)$, $\overset{u}{\overrightarrow{t}} = t$.*

What is worse, it is possible that $\overset{ua}{\overrightarrow{t}} \neq t$. Thus if we explicitly create a node for every $s \in \mathit{Suffix}_D(u) - \mathit{Suffix}_D(\mathit{lrs}_D(ua) \cdot a^{-1}) - [u]^E_u$ for each $u \in \mathit{Prefix}(w)$, it can take $O(n^2)$ time in total. To avoid this, we 'postpone' creating such a node until we get the first symbol a satisfying $a \neq b = u[i+1]$ for any $i \in \mathit{Endpos}_u(s) \cap (\mathit{Wordpos}_D(u) \oplus |s| \ominus 1)$. (see Figure 3.) This timing coincides with when we insert a new a-labeled edge to the old sink node as mentioned above.

Due to Proposition 7, the new node $[s]^E_{ua}$ can contain more than one string from $\mathit{Suffix}_D(s)$. The equivalence test can be performed according to Lemma 21 given below. Before that, we show Lemma 20 which supports Lemma 21.

Lemma 20. *Let $w \in D^+$, $u \in \mathit{Prefix}(w)$, and $x \in \mathit{Prefix}(\mathit{Suffix}_D(u))$. Let $\overset{u}{\overrightarrow{x}} = z_1$. For any $i \in \mathit{Begpos}_{z_1}(x)$ such that $i > 1$, we have $z_1[i-1] \neq \#$. Similarly, let $\overset{u}{\overleftarrow{x}} = z_2$. For any $j \in \mathit{Begpos}_{z_2}(x)$ such that $j \leq |z_2| - |x|$, we have $z_2[j-1] \neq \#$.*

Proof. Assume for contrary that $z_1[i-1] = \#$. Then, $\mathit{Begpos}_u(x) \cap \mathit{Wordpos}_D(u) = (\mathit{Begpos}_u(z_1) \cup (\mathit{Begpos}_u(z_1) \oplus (i-1)) \cap \mathit{Wordpos}_D(u)$. Since $i > 1$, $\mathit{Begpos}_u(z_1) \oplus (i-1) \neq \mathit{Begpos}_u(z_1)$. Moreover, $\mathit{Begpos}_u(z_1) \oplus (i-1) \subseteq \mathit{Wordpos}_D(u)$. Thus, $(\mathit{Begpos}_u(z_1) \cup (\mathit{Begpos}_u(z_1) \oplus (i-1)) \cap \mathit{Wordpos}_D(u) \neq \mathit{Begpos}_u(z_1) \cap \mathit{Wordpos}_D(u)$, which implies that $\mathit{Begpos}_u(x) \cap \mathit{Wordpos}_D(u) \neq \mathit{Begpos}_u(z_1) \cap \mathit{Wordpos}_D(u)$. However, this contradicts with $x \equiv^B_u z_1$. Similar arguments hold for $\overset{u}{\overleftarrow{x}} = z_2$. □

Lemma 21. *Let $w \in D^+$ and $u \in \mathit{Prefix}(w)$. For any $x, y \in \mathit{Prefix}(\mathit{Suffix}_D(u))$ such that $y \in \mathit{Suffix}_D(x)$,*

$$x \equiv^E_u y \Leftrightarrow [\overset{u}{\overrightarrow{x}}]^E_u = [\overset{u}{\overrightarrow{y}}]^E_u.$$

Proof. If $x \equiv^E_u y$, then $\overset{u}{\overleftarrow{x}} = \overset{u}{\overleftarrow{y}}$. By Proposition 9, $\overset{u}{\overleftarrow{(\overset{u}{\overrightarrow{x}})}} = \overset{u}{\overrightarrow{(\overset{u}{\overleftarrow{x}})}}$ and $\overset{u}{\overleftarrow{(\overset{u}{\overrightarrow{y}})}} = \overset{u}{\overrightarrow{(\overset{u}{\overleftarrow{y}})}}$. Thus we get $\overset{u}{\overleftarrow{(\overset{u}{\overrightarrow{x}})}} = \overset{u}{\overleftarrow{(\overset{u}{\overrightarrow{y}})}}$, which implies that $[\overset{u}{\overrightarrow{x}}]^E_u = [\overset{u}{\overrightarrow{y}}]^E_u$.

Now suppose $[\overset{u}{\overrightarrow{x}}]^E_u = [\overset{u}{\overrightarrow{y}}]^E_u$. Let $\overset{u}{\overrightarrow{x}} = x\alpha$ and $\overset{u}{\overleftarrow{x}} = \beta x$ for some strings $\alpha, \beta \in (\Sigma \cup \{\#\})^*$. Let $z = \overset{u}{\overleftrightarrow{x}}$. Then $z = \beta x \alpha$ by definition. Since $y \in \mathit{Suffix}_D(x)$, there

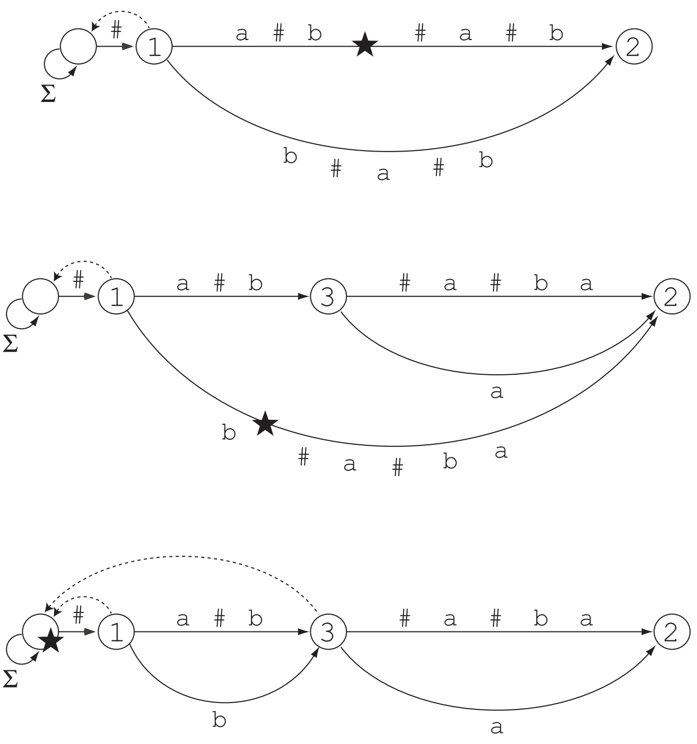

Figure 4. A new node is created in the update of $SCDAWG_D(u)$ to $SCDAWG_D(u\mathtt{a})$, where $u = \mathtt{a\#b\#a\#b}$. The stars represent the location from which we check a transitivity with a new symbol $\mathtt{a}$. When the transitivity check failed on an edge (namely, not right on a node), we divide the edge into two at the check point (where the star lies) and create a new node there (Node 3 in this figure) together with a new edge labeled with the new character leading to the sink node. Assume that the next transitivity check point also lies on an edge. We apply Lemma 21 and if it is the case, we 'shorten' the edge till the check point and merge this shortened edge to the above created node, as seen in the conversion from the second graph to the third one of this figure.

exists some string $\gamma \in (\Sigma \cup \{\#\})^*$ such that $\gamma y = x$. Because $\overset{u}{\overleftrightarrow{x}} = \overset{u}{\overleftrightarrow{y}}$, $z = \beta\gamma y\alpha$. By Lemma 20, $\overset{u}{\overrightarrow{y}} = y\alpha$ and $\overset{u}{\overleftarrow{y}} = \beta\gamma y$. Hence,

$$\begin{aligned}
& \mathit{Endpos}_u(x) \cap (\mathit{Wordpos}_D(u) \oplus |x| \ominus 1) \\
={} & \mathit{Endpos}_u(\beta x) \cap (\mathit{Wordpos}_D(u) \oplus |\beta x| \ominus 1) \\
={} & \mathit{Endpos}_u(\beta\gamma y) \cap (\mathit{Wordpos}_D(u) \oplus |\beta\gamma y| \ominus 1) \\
={} & \mathit{Endpos}_u(y) \cap (\mathit{Wordpos}_D(u) \oplus |y| \ominus 1),
\end{aligned}$$

which implies that $x \equiv^E_u y$. □

Figure 4 displays how a new node is created.

Splitting a node. Lastly, we remark that there is a possibility that a certain node of $SCDAWG_D(u)$ can be split into two nodes in $SCDAWG_D(ua)$, as shown in the following lemma that has inherently been shown in [9].

Lemma 22. *Let $w \in D^+$, and let $u, ua \in \mathit{Prefix}(w)$ with $a \in \Sigma \cup \{\#\}$. Let $z = \mathit{lrs}_D(ua)$. Then, for any $x \in \mathit{Prefix}(\mathit{Suffix}_D(u))$, we have*

$$[x]^E_u = \begin{cases} [\overset{u}{\overleftrightarrow{x}}]^E_{ua} \cup [z]^E_{ua}, & \textit{if } z \in [x]^E_u \textit{ and } z \neq \overset{u}{\overleftrightarrow{x}}; \\ [x]^E_{ua}, & \textit{otherwise.} \end{cases}$$

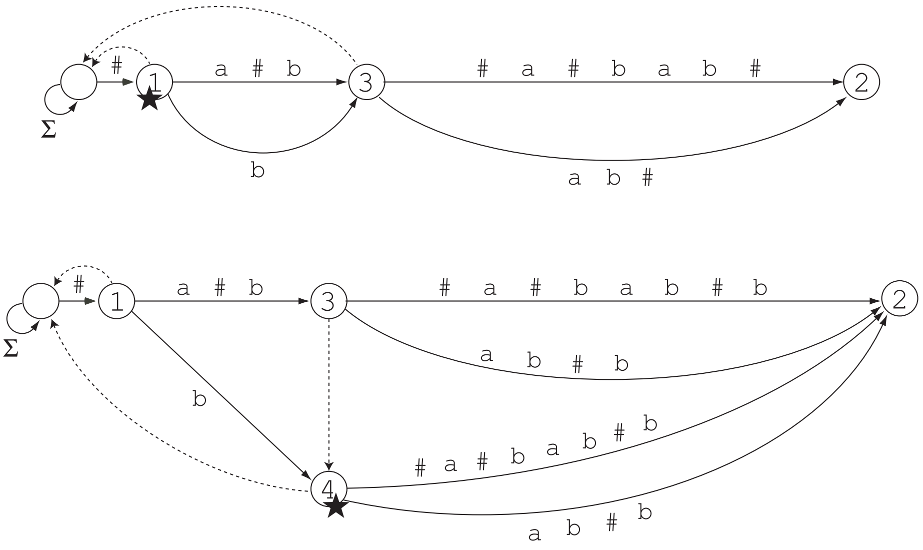

Figure 5. Illustration for node splitting. Node 3 is split in the update of $\mathit{SDAWG}_D(u)$ to $\mathit{SCDAWG}_D(u\mathtt{b})$ with $u = \mathtt{a\#b\#a\#bab\#}$. The stars represent the location from which we check a transitivity with a new symbol.

To node $[x]^E_u$ such that $z \in [x]^E_u$, we examine whether $z = \overset{u}{\overleftrightarrow{x}}$ or not by checking the length of $\overset{u}{\overleftrightarrow{x}}$ and z, as follows. Consider edge $([y]^E_u, \alpha, [x]^E_u)$ such that $\overset{u}{\overleftrightarrow{y}} \cdot \alpha = z$. Then, it is easy to see that

$$z = \overset{u}{\overleftrightarrow{x}} \Leftrightarrow |\overset{u}{\overleftrightarrow{y}} \cdot \alpha| = |\overset{u}{\overleftrightarrow{x}}| \Leftrightarrow |\overset{u}{\overleftrightarrow{y}}| + |\alpha| = |\overset{u}{\overleftrightarrow{x}}| \Leftrightarrow \mathit{length}([y]^E_u) + |\alpha| = \mathit{length}([x]^E_u).$$

Setting the length of the initial state q_s of M_D to be -1, no contradiction occurs even in case that $z = \varepsilon$.

See Figure 5 for a concrete example of node splitting.

Pseudo Code. A pseudo code of our on-line algorithm to build SCDAWGs is shown in Figures 6 and 7. Any edge label $\alpha \in (\Sigma \cup \{\#\})^+$ is implemented as an ordered pair (i, j) of positions such that $u[i..j] = \alpha$, in order to implement the SCDAWG with $O(k)$ space. To neglect to extend the existing in-coming edges of *sink* node (refer to Proposition 17), we implement by (i, ∞) the label of any edge leading to the sink node. This is the same idea as in [14].

For each $i = 1, \dots, n$, we call function *Update* which converts $\mathit{SCDAWG}_D(w[1..i-1])$ to $\mathit{SCDAWG}_D(w[1..i])$ and returns the location of $\mathit{lrs}_D(w[1..i])$ by pair (s, k). Here, s is the lowest node in the path that spells out $\mathit{lrs}_D(w[1..i])$ from the source node. Let $\mathit{lrs}_D(w[1..i]) = xy$ such that x belongs to the equivalence class of node s and y is the rest. Then, k is the integer such that $w[k : i] = y$.

Function *CheckEndPoint* returns **true** if the location indicated by triple $(s, (k, i))$ corresponds to $\mathit{lrs}_D(w[1..i])$, and **false** otherwise. Due to Proposition 17, in the **while** loop of function *Update*, we create new edges $(r, (i, \infty), \mathit{sink})$. This is continuously done until finding $\mathit{lrs}_D(w[1..i])$, according to Proposition 18. Consider the case that $(s, (k, i))$ lies on an edge, and that we have created a new node r in the first **else** condition of the **while** loop. Due to Proposition 21, when the second **if** condition is satisfied, we shorten the edge and redirect it to node r . Note that, since s' is initially set to **nil**, this second **if** condition can be satisfied only after the **else** condition is

```
 **Input:**    $w = w[1..n] \in D^+$ and $M_D$ with initial state $q_s$ and final state $q_f$.
**Output:**    $SCDAWG_D(w)$.
{
    /* We assume $\Sigma = \{w[-1], w[-2], \ldots, w[-m]\}$ */.
    /* Replace the edge labels of $M_D$ with appropriate integer pairs */.
    $length(q_f) = 0$;    $length(q_s) = -1$;    $length(sink) = \infty$;
    $source = q_f$;    $link(source) = q_s$;    $link(sink) =$ **nil**;
    $(s, k) = (source, 1)$;
    **for** $(i = 1; i \leq n; i{+}{+})$  $(s, k) = Update(s, (k, i))$;
}

 **(node,integer)-pair** $Update(s, (k, i))$ {
    $oldr =$ **nil**; $s' =$ **nil**;
    **while** $(CheckEndPoint(s, (k, i-1), w[i]) ==$ **false**) {
        **if** $(k \leq i-1)$ {  /* $(s, (k, i-1))$ is implicit. */
            **if** $(s' == Extension(s, (k, p-1)))$ {
                let $(s, (k', p'), s')$ be the $w[k]$-edge from $s$;
                replace the edge by edge $(s, (k', k'+p-k-1), r)$;
                $(s, k) = Canonize(link(s), (k, p-1))$;
                **continue**;
            }
            **else** {
                $s' = Extension(s, (k, p-1))$;
                $r = CreateNode(s, (k, p-1))$;
            }
        } **else** $r = s$;    /* $(s, (k, i-1))$ is explicit. */
        create new edge $(r, (i, \infty), sink)$;
        **if** $(oldr \neq$ **nil**) $link(oldr) = r$;
        $oldr = r$;
        $(s, k) = Canonize(link(s), (k, i-1))$;
    }
    **if** $(oldr \neq$ **nil**) $link(oldr) = s$;
    **return** $SplitNode(s, (k, i))$;
}
```

Figure 6. Main routine and function *Update* of our SCDAWG construction algorithm. For any node s, $link(s)$ denotes the node to which the suffix link of s goes. By ‘implicit’ we mean that the location is on an edge, and by ‘explicit’ we mean that it is on a node.

```
boolean CheckEndPoint($s, (k, p), c$) {
    if ($k \leq p$) {    /* $(s, (k, p))$ is implicit. */
        let $(s, (k', p'), s')$ be the $w[k]$-edge from $s$;
        return ($c == w[k' + p - k + 1]$);
    } else return (there is a $c$-edge from $s$);
}

node Extension($s, (k, p)$) {
    if ($k > p$) return $s$;    /* $(s, (k, p))$ is explicit. */
    find the $w[k]$-edge $(s, (k', p'), s')$ from $s$;
    return $s'$;
}

(node,integer)-pair Canonize($s, (k, p)$) {
    if ($k > p$) return $(s, k)$;    /* $(s, (k, p))$ is explicit. */
    find the $w[k]$-edge $(s, (k', p', s')$ from $s$;
    while ($p' - k' \leq p - k$) {
        $k = k + p' - k' + 1$;
        $s = s'$;
        if ($k \leq p$) find the $w[k]$-edge $(s, (k', p'), s')$ from $s$;
    }
    return $(s, k)$;
}

node CreateNode($s, (k, p)$) {
    let $(s, (k', p'), s')$ be the $w[k]$-edge from $s$;
    create new node $r$;
    replace the edge by edges $(s, (k', k' + p - k), r)$ and $(r, (k' + p - k + 1, p'), s')$;
    $length(r) = length(s) + (p - k + 1)$;
    return $r$;
}

(node,integer)-pair SplitNode($s, (k, p)$) {
    $(s', k') = Canonize(s, (k, p))$;
    if ($k' \leq p$) return $(s', k')$;    /* $(s', (k', p))$ is implicit. */
    /* $(s', (k', p))$ is explicit. */
    if ($length(s') == length(s) + (p - k + 1)$) return $(s', k')$;
    create node $r'$ as a duplication of $s'$ with the out-going edges;
    $link(r') = link(s')$;    $link(s') = r'$;
    $length(r') = length(s) + (p - k + 1)$;
    do {
        replace the $w[k]$-edge from $s$ to $s'$ by edge $(s, (k, p), r')$;
        $(s, k) = Canonize(link(s), (k, p - 1))$;
    } while ($(s', k') = Canonize(s, (k, p))$);
    return $(r', p + 1)$;
}
```

Figure 7. The other functions of our on-line SCDAWG construction algorithm. All these functions are identical to those in [7].

once satisfied and s' gets a non-**nil** value. After creating new edge $(r, (i, \infty), sink)$, we traverse the suffix link of node s to find $lrs_D(w[1..i])$.

After the insertion of all the new edges, we call function *SplitNode* that splits the node corresponding to $lrs_D(w[1..i])$ into two nodes, when needed. This operation is due to Lemma 22.

We remark that the only difference between our algorithm and the on-line algorithm of [7] for constructing normal CDAWGs is the initialization steps of the main routine where we set the source of the SCDAWG to the final state q_f of M_D and the suffix link of the source to the initial state q_s of M_D. These simple modifications make the proposed algorithm construct SCDAWGs together with their suffix links.

For the correctness of the algorithm, we attach an end-marker \$ to any input string $w \in D^+$, which appears nowhere in w. Possible problems that may be caused by the 'delay' of creating new nodes, can be cleared by this end-marker, since \$ appears nowhere in w.

Theorem 23. *For any string $w \in D^+$, the algorithm of Figures 6 and 7 correctly constructs $SCDAWG_D(w\$)$.*

Now the only remaining matter is the time complexity of the algorithm. The following theorem can be proven by the same idea as the linearity proof of the normal CDAWG construction algorithm in [7].

Theorem 24. *For any $w \in D^+$ such that $w = w_1 \cdots w_k$ and $|w| = n$, the algorithm of Figures 6 and 7 runs in $O(n)$ time using $O(k)$ space.*

5 Conclusions and Open Problems

In this paper we introduced a new text indexing structure, sparse compact directed acyclic word graphs (SCDAWGs) for strings $w = w_1 \cdots w_k$ over dictionary $D = \Sigma^*\#$. We showed that SCDAWGs require only $O(k)$ space and are strictly smaller than sparse (word) suffix trees. Furthermore, we presented an on-line algorithm that builds SCDAWGs in $O(n)$ time, where $n = |w|$. The proposed algorithm correctly builds SCDAWGs with the help of the minimum DFA M_D accepting D, and the tailored suffix links. SCDAWGs are expected to become a space-economical alternative to sparse suffix trees in application areas such as natural language processing, biological sequence analysis, etc.

Here are some open problems regarding sparse text indexing structures:

1. Exact numbers of nodes and edges of SCDAWGs. Being a tree with k leaves and only branching internal nodes, any sparse suffix tree can have at most $2k-1$ nodes and $2k-2$ edges. Thus, it is guaranteed that any SCDAWG has less nodes and edges than these.
2. Would it be possible to construct *sparse suffix arrays* efficiently? Sparse suffix arrays can be obtained from the leaves of the corresponding sparse suffix trees, but is it possible to build sparse suffix arrays directly, and in $O(n)$ time with $O(k)$ space?

References

[1] A. Andersson, N. J. Larsson, and K. Swanson: *Suffix trees on words.* Algorithmica, 23(3) 1999, pp. 246–260.

[2] H. Bannai, S. Inenaga, A. Shinohara, M. Takeda, and S. Miyano: *Efficiently finding regulatory elements using correlation with gene expression.* Journal of Bioinformatics and Computational Biology, 2(2) 2004, pp. 273–288.

[3] A. Blumer, J. Blumer, D. Haussler, A. Ehrenfeucht, M. T. Chen, and J. Seiferas: *The smallest automaton recognizing the subwords of a text.* Theoretical Computer Science, 40 1985, pp. 31–55.

[4] A. Blumer, J. Blumer, D. Haussler, R. McConnell, and A. Ehrenfeucht: *Complete inverted files for efficient text retrieval and analysis.* Journal of the ACM, 34(3) 1987, pp. 578–595.

[5] B. Dorohonceanu and C. G. Nevill-Manning: *Accelerating protein classification using suffix trees*, in Proc. 8th International Conference on Intelligent Systems for Molecular Biology (ISMB'00), AAAI Press, 2000, pp. 128–133.

[6] S. Inenaga, T. Funamoto, M. Takeda, and A. Shinohara: *Linear-time off-line text compression by longest-first substitution*, in Proc. 10th International Symp. on String Processing and Information Retrieval (SPIRE'03), vol. 2857 of Lecture Notes in Computer Science, Springer-Verlag, 2003, pp. 137–152.

[7] S. Inenaga, H. Hoshino, A. Shinohara, M. Takeda, S. Arikawa, G. Mauri, and G. Pavesi: *On-line construction of compact directed acyclic word graphs.* Discrete Applied Mathematics, 146(2) 2005, pp. 156–179.

[8] S. Inenaga and M. Takeda: *On-line linear-time construction of word suffix trees*, in Proc. 17th Ann. Symp. on Combinatorial Pattern Matching (CPM'06), vol. 4009 of Lecture Notes in Computer Science, Springer-Verlag, 2006, pp. 60–71.

[9] S. Inenaga and M. Takeda: *Sparse directed acyclic word graphs*, in Proc. 13th International Symp. on String Processing and Information Retrieval (SPIRE'06), Lecture Notes in Computer Science, Springer-Verlag, 2006, To appear.

[10] N. J. Larsson: *Extended application of suffix trees to data compression*, in Proc. Data Compression Conference '96 (DCC'96), IEEE Computer Society, 1996, pp. 190–199.

[11] L. Marsan and M.-F. Sagot: *Extracting structured motifs using a suffix tree - algorithms and application to promoter consensus identification*, in Proc. 4th Annual International Conference on Computational Molecular Biology (RECOMB'00), ACM, 2000, pp. 210–219.

[12] J. C. Na, A. Apostolico, C. S. Iliopoulos, and K. Park: *Truncated suffix trees and their application to data compression.* Theoretical Computer Science, 304(1–3) 2003, pp. 87–101.

[13] D. Revuz: *Minimisation of acyclic deterministic automata in linear time.* Theoretical Computer Science, 92(1) 1992, pp. 181–189.

[14] E. Ukkonen: *On-line construction of suffix trees.* Algorithmica, 14(3) 1995, pp. 249–260.