On-Line Construction of Symmetric Compact Directed Acyclic Word Graphs

Shunsuke Inenaga† Hiromasa Hoshino† Ayumi Shinohara†
Masayuki Takeda†,‡ Setsuo Arikawa†
† Department of Informatics, Kyushu University 33, Fukuoka 812-8581, Japan
‡ PRESTO, Japan Science and Technology Corporation (JST)
E-mail: {s-ine,hoshino,ayumi,takeda,arikawa}@i.kyushu-u.ac.jp

Abstract

The Compact Directed Acyclic Word Graph (CDAWG) is a space efficient data structure that supports indices of a string. The Symmetric Directed Acyclic Word Graph (SCDAWG) for a string w is a dual structure that supports indices of both w and the reverse of w simultaneously. Blumer et al. gave the first algorithm to construct an SCDAWG from a given string, that works in an off-line manner. In this paper, we show an on-line algorithm that constructs an SCDAWG from a given string directly.

1 Introduction

A Directed Acyclic Word Graph (DAWG) is the smallest finite state automaton that recognizes all the suffixes of a given string [1]. DAWGs are involved in several combinatorial algorithms on strings, because they serve as indices of strings, as well as other indexing structures like suffix tries, suffix trees, and suffix arrays (see eg. [2, 4, 8]). All of these indexing structures except for suffix trie can be constructed in linear time with respect to the length of a given string, and the space requirements are also linear. The hidden constants behind the big-O notation of space complexity are critical in practice, and much attention has recently been paid to reduce these constants.

Blumer et al. [2] first introduced the *Compact Directed Acyclic Word Graph* (CDAWG), a space efficient variant of a DAWG. A CDAWG can be obtained by not only compacting the corresponding DAWG, but also minimizing the corresponding suffix tree [5]. Blumer et al. gave a linear time algorithm for constructing the CDAWG for a given string, which first builds the DAWG for the string and then shrinks it to the CDAWG. Later, Crochemore and Vérin [5] developed the first algorithm to construct CDAWGs *directly*, that is, without constructing DAWGs or suffix trees as intermediates. Their algorithm allows us to save time and space requirements simultaneously, since it has been proven that the size of CDAWGs is strictly smaller than those of DAWGs and suffix trees [2, 5]. Recently, we proposed an *on-line* algorithm that directly constructs CDAWGs from given strings [9]. The algorithm is based on Ukkonen's on-line suffix tree construction algorithm given in [15], while the one Crochemore and Vérin gave is based on McCreight's off-line suffix tree construction algorithm [11].

A kind of failure transition, *suffix link*, is often used for efficient constructions of indexing structures such as suffix tries, suffix trees, DAWGs, and CDAWGs [16, 11, 1, 2, 15, 5, 9]. An interesting fact is that, for any string w, the suffix links of $STrie(w)$ form $STrie(w^{\rm rev})$ [6], where $w^{\rm rev}$ denotes the reversal of w. A DAWG also has a similar property, that is, the suffix links of the $DAWG(w)$ compose $STree(w^{\rm rev})$ [3]. However, this duality is damaged in case of suffix trees. Namely, the suffix links of $STree(w)$ do not form a structure supporting indices of $w^{\rm rev}$. However, the set of suffix links of $STree(w)$ corresponds to a subset of the set of edges of $DAWG(w^{\rm rev})$ [4]. In order to obtain the duality on suffix trees, the *affix tree* is developed by Stoye [12, 13]. Affix trees are the modification of suffix trees so that the suffix links of $ATree(w)$ form $ATree(w^{\rm rev})$ (see Fig. 5). Stoye could not prove his on-line algorithm for constructing affix trees runs in linear time, but Maaß [10] later succeeded to improve it so as to run in lin-

ear time. Meanwhile, Blumer et al. [2] showed that the nodes of a CDAWG are invariant under reversal: the nodes of the CDAWG for a string w exactly correspond to those of the CDAWG for w^{rev}, which they call the Symmetric Compact Directed Acyclic Word Graph (SCDAWG) for w (see Fig. 4, right).

In [15], Ukkonen gave intuitive and excellent on-line algorithms for the construction of $STrie(w)$ and $STree(w)$. Since the suffix links of $STrie(w)$ are equal to the edges of $STrie(w^{rev})$, it turns out that $STrie(w)$ and $STrie(w^{rev})$ sharing the same nodes can be simultaneously built on-line, scanning w from left to right. Also, as the algorithm to construct DAWGs which Blumer et al. gave in [1] is on-line, it results in that their algorithm builds $DAWG(w)$ and $STree(w^{rev})$ at the same time, in on-line (left to right) fashion. Moreover, the fact is that the first algorithm that constructs suffix trees, given by Weiner in [16], becomes more interesting when considered as an on-line algorithm. His algorithm builds the suffix tree for a string w by appending the suffixes of w to the current suffix tree in increasing order. In other words, his algorithm builds $STree(w)$ on-line, *right to left*. In addition to that, his algorithm can be modified so as to create the edges of the DAWG for w^{rev} at the same time [5]. It implies that his algorithm also simultaneously constructs $DAWG(w)$ together with $STree(w^{rev})$ on-line, left to right.

In this paper, we first give an algorithm that simultaneously builds $STree(w)$ with $DAWG(w^{rev})$ on-line, left to right. This algorithm constructs $STree(w)$ in the same way as the Ukkonen algorithm does, while computing the *shortest extension links* (sext links) that form $DAWG(w^{rev})$ at the same time. Moreover, we show an algorithm that *directly* constructs $SCDAWG(w)$ *on-line*, left to right. It builds $CDAWG(w)$ similarly to the algorithm we developed in [9], and computes the sext links that are equal to the edges of $CDAWG(w^{rev})$.

From a practical point of view, SCDAWGs and affix trees have essentially the same range of applications. However, the number of nodes in $SCDAWG(w)$ is much smaller than that of $ATree(w)$, although both are linear with respect to the length of a given string w. In fact, an inequality comparing the number of nodes

$$\begin{aligned} & |SCDAWG(w)| \\ \leq\ & \min\{|STree(w)|, |STree(w^{rev})|\} \\ \leq\ & \max\{|STree(w)|, |STree(w^{rev})|\} \\ \leq\ & |ATree(w)| \end{aligned}$$

holds for any string w. This is because, intuitively, the set of nodes in $SCDAWG(w)$ is the *intersection* of those in $STree(w)$ and $STree(w^{rev})$, while the set of nodes in $ATree(w)$ is the *union* of them. Therefore, SCDAWGs save space considerably compared to affix trees. Moreover, not only a CDAWG is attractive as indexing structure, but also the underlying equivalence relation is useful in data mining or machine discovery from textual databases. Actually, the equivalence relation plays a central role in supporting human experts who involved in evaluation/interpretation task for mined expressions from anthologies of classical Japanese poems [14].

The rest of this paper is organized as follows. In Section 2, we introduce some notions and notation, and define suffix tries, suffix trees, DAWGs, and CDAWGs in terms of the equivalence relations over strings. It gives a unified view for these data structures. Moreover, we define some bidirectional indexing structures, including SCDAWGs, from the viewpoint of the duality. In Section 3, we give an on-line linear time algorithm to construct $STree(w)$ with $DAWG(w^{rev})$ simultaneously. Later on, an algorithm that builds $SCDAWG(w)$ on-line in linear time is shown in Section 4. We conclude in Section 5.

2 Preliminaries

2.1 Notation

Let Σ be a finite alphabet. An element of Σ^* is called a *string*. Strings x, y, and z are said to be a *prefix*, *factor*, and *suffix* of string $u = xyz$, respectively. The sets of prefixes, factors, and suffixes of a string w are denoted by $Prefix(w)$, $Factor(w)$, and $Suffix(w)$, respectively. The length of a string u is denoted by $|u|$. The empty string is denoted by ε, that is, $|\varepsilon| = 0$. Let $\Sigma^+ = \Sigma^* - \{\varepsilon\}$. The ith symbol of a string w is denoted by $w[i]$ for $1 \leq i \leq |w|$, and the factor of a string w that begins at position i and ends at position j is denoted by $w[i:j]$ for $1 \leq i \leq j \leq |w|$. For convenience, let $w[i:j] = \varepsilon$ for $j < i$. For a string w, we denote by w^{rev} the reversed string of w, that is, $w^{rev} = w[n] \ldots w[2]w[1]$. For a set S of strings, let $|S|$ denote the cardinality of S.

2.2 Equivalence Relations on Strings

For strings $x, y \in \Sigma^*$, we write as $x \equiv_w^L y$ (resp. $x \equiv_w^R y$) if the sets of positions in w at which x and y begin (resp. end) are identical. The equivalence class of a string $x \in \Sigma^*$ with respect to

$\equiv_w^L$ (resp. $\equiv_w^R$) is denoted by $[x]_w^L$ (resp. $[x]_w^R$). For instance, if $w = \texttt{baggage}$, then $[\texttt{a}]_w^L = \{\texttt{a}, \texttt{ag}\}$, $[\texttt{ga}]_w^L = \{\texttt{ga}, \texttt{gag}, \texttt{gage}\}$, $[\texttt{a}]_w^R = \{\texttt{a}\}$, $[\texttt{ga}]_w^R = \{\texttt{ga}, \texttt{gga}, \texttt{agga}, \texttt{bagga}\}$, and so on.

Note that all strings that are not in $Factor(w)$ form one equivalence class under $\equiv_w^L$ ($\equiv_w^R$). This equivalence class is called the *degenerate* class. All other classes are called *non-degenerate*. It follows from the definition of $\equiv_w^L$ that if two factors x and y of w are in a single equivalent class under $\equiv_w^L$, then either x is a prefix of y, or vice versa. Therefore, each equivalence class in $\equiv_w^L$ other than the degenerate class has a unique longest member. Similar discussion holds for $\equiv_w^R$.

For any factor x of a string $w \in \Sigma^*$, let $\overrightarrow{x}^w$ and $\overleftarrow{x}^w$ denote the unique longest members of $[x]_w^L$ and $[x]_w^R$, respectively. We call $\overrightarrow{x}^w$ (resp. $\overleftarrow{x}^w$) the *representative* of $[x]_w^L$ (resp. $[x]_w^R$). In the running example, $\overrightarrow{\texttt{a}}^w = \texttt{ag}$, $\overrightarrow{\texttt{b}}^w = \texttt{baggage}$, $\overleftarrow{\texttt{a}}^w = \texttt{a}$, $\overleftarrow{\texttt{gg}}^w = \texttt{bagg}$, and so on. Moreover, let $\overleftrightarrow{x}^w$ denote the string $\alpha x \beta$ such that $\overleftarrow{x}^w = \alpha x$ and $\overrightarrow{x}^w = x\beta$, where $\alpha, \beta \in \Sigma^*$. In the running example, $\overleftrightarrow{\texttt{a}}^w = \texttt{ag}$, and $\overleftrightarrow{\texttt{gg}}^w = \texttt{baggage}$. For any strings $x, y \in Factor(w)$, $x \equiv_w y$ if and only if $\overleftarrow{x}^w = \overleftarrow{y}^w$.

2.3 Suffix Tries, Suffix Trees, DAWGs, and CDAWGs

We here recall four indexing structures, the suffix trie, the suffix tree, the DAWG, and the CDAWG for a string $w \in \Sigma^*$, all of which represent every string $x \in Factor(w)$. They are denoted by $STrie(w)$, $STree(w)$, $DAWG(w)$, and $CDAWG(w)$, respectively. We define them as edge-labeled graphs (V, E) with $E \subseteq V \times \Sigma^+ \times V$ where the second component of each edge represents its label. The definitions of $STrie(w)$, $STree(w)$, $DAWG(w)$, and $CDAWG(w)$ are given in Fig. 1.

One can regard $STree(w)$ as the compacted version of $STrie(w)$ with "$\overrightarrow{x}^w$ operation". Similarly, $DAWG(w)$ can be seen as the minimized version of $STrie(w)$ with "$[x]_w^R$ operation". These different operations cause the difference between suffix trees and DAWGs: $\overrightarrow{x}^w$ refers to a single string, the representative of $[x]_w^L$, while more than one string can belong to $[x]_w^R$. That means, a node of $STree(w)$ represents just one string in $Factor(w)$, but that of $DAWG(w)$ can represent more than one. In other words, two or more edges may point to the same node in $DAWG(w)$. $CDAWG(w)$ is obtained both by minimizing $STree(w)$ with "$[x]_w^R$ operation" and by compacting $DAWG(w)$ with "$\overrightarrow{x}^w$ operation".

The nodes of $STrie(w)$ and $STree(w)$ corresponding to ε are called the *root* nodes. On a common assumption that a string w ends with an *end-marker* occurring only at the end of w, every string $x \in Suffix(w)$ except ε is associated with some *leaf* node both in $STrie(w)$ and in $STree(w)$. In other words, both $STrie(w)$ and $STree(w)$ have $|Suffix(w)| - 1$ leaf nodes. An end-marker is denoted by \$. From here on, let us make the end-marker assumption keep on holding until the end of this paper. The nodes corresponding to ε in $DAWG(w)$ and $CDAWG(w)$ are called the *initial* nodes. Both of them have the *final* node with which every string $x \in Suffix(w)$ but ε is associated.

2.4 Bidirectional Index Structures

If an index structure represents all the strings not only in $Factor(w)$ but also in $Factor(w^{\mathrm{rev}})$, let us call it a *bidirectional* index structure for string w. We define such a structure as a graph with two kinds of edges: the ones for a string w, and the other for w^{rev}.

Giegerich and Kurtz [6] observed that $STrie(w)$ and $STrie(w^{\mathrm{rev}})$ are dual in the sense that they share the same nodes. We refer this bidirectional index structure as "$STrie(w)$ with $STrie(w^{\mathrm{rev}})$". The formal definition is given as Definition 5 in Fig. 2

The duality of $STree(w)$ and $DAWG(w^{\mathrm{rev}})$, which was pointed out in [3, 4], is shown in Definition 6 of Fig. 2. Let $V' = \{[x]_w^L \mid x \in Factor(w)\}$. It is easy to see that there is a trivial one-to-one correspondence between V of Definition 6 and V'. Using this correspondence, we can identify $E_{R\to L}$ of Definition 6 with

$$\left\{ ([x]_w^L, a, [ax]_w^L) \,\middle|\, \begin{array}{l} x, ax \in Factor(w) \\ \text{and } a \in \Sigma \end{array} \right\} = \left\{ ([y]_{w^{\mathrm{rev}}}^R, a, [ya]_{w^{\mathrm{rev}}}^R) \,\middle|\, \begin{array}{l} y, ya \in Factor(w^{\mathrm{rev}}) \\ \text{and } a \in \Sigma \end{array} \right\},$$

which is equivalent to the definition of $DAWG(w^{\mathrm{rev}})$.

The edges $E_{R\to L}$ of Definition 6 are the so-called *shortest extension links* (*sext links*) of $STree(w)$, which were introduced by Crochemore and Rytter in [4]. Moreover, a part of the reversed sext links are known as *suffix links* that play a key role in time efficient construction of suffix trees, DAWGs, and CDAWGs. Formally, the suffix links

Definition 1 *$STrie(w)$ is the tree (V, E) such that*

$$\begin{aligned} V &= \{x \mid x \in \mathit{Factor}(w)\}, \\ E &= \{(x, a, xa) \mid x, xa \in \mathit{Factor}(w) \text{ and } a \in \Sigma\}. \end{aligned}$$

Definition 2 *$STree(w)$ is the tree (V, E) such that*

$$\begin{aligned} V &= \{\overset{w}{\overrightarrow{x}} \mid x \in \mathit{Factor}(w)\}, \\ E &= \{(\overset{w}{\overrightarrow{x}}, a\beta, \overset{w}{\overrightarrow{xa}}) \mid x, xa \in \mathit{Factor}(w),\ a \in \Sigma,\ \beta \in \Sigma^*,\ \overset{w}{\overrightarrow{xa}} = xa\beta, \text{ and } \overset{w}{\overrightarrow{x}} \neq \overset{w}{\overrightarrow{xa}}\}. \end{aligned}$$

Definition 3 *$DAWG(w)$ is the dag (V, E) such that*

$$\begin{aligned} V &= \{[x]_w^R \mid x \in \mathit{Factor}(w)\}, \\ E &= \{([x]_w^R, a, [xa]_w^R) \mid x, xa \in \mathit{Factor}(w) \text{ and } a \in \Sigma\}. \end{aligned}$$

Definition 4 *$CDAWG(w)$ is the dag (V, E) such that*

$$\begin{aligned} V &= \{[\overset{w}{\overrightarrow{x}}]_w^R \mid x \in \mathit{Factor}(w)\} \simeq \{\overset{w}{\overleftarrow{x}} \mid x \in \mathit{Factor}(w)\}, \\ E &= \{([\overset{w}{\overrightarrow{x}}]_w^R, a\beta, [\overset{w}{\overrightarrow{xa}}]_w^R) \mid x, xa \in \mathit{Factor}(w),\ a \in \Sigma,\ \beta \in \Sigma^*,\ \overset{w}{\overrightarrow{xa}} = xa\beta, \text{ and } \overset{w}{\overrightarrow{x}} \neq \overset{w}{\overrightarrow{xa}}\} \\ &\simeq \{(\overset{w}{\overleftarrow{x}}, a\beta, \overset{w}{\overleftarrow{xa}}) \mid x, xa \in \mathit{Factor}(w),\ a \in \Sigma,\ \beta \in \Sigma^*,\ \overset{w}{\overrightarrow{xa}} = xa\beta, \text{ and } \overset{w}{\overrightarrow{x}} \neq \overset{w}{\overrightarrow{xa}}\}. \end{aligned}$$

Figure 1. The definitions of the indexing structures $STrie(w)$, $STree(w)$, $DAWG(w)$, and $CDAWG(w)$.

Definition 5 *$STrie(w)$ with $STrie(w^{rev})$ is the bidirectional tree $(V, E_{L\to R}, E_{R\to L})$ such that*

$$\begin{aligned} V &= \{x \mid x \in \mathit{Factor}(w)\}, \\ E_{L\to R} &= \{(x, a, xa) \mid x, xa \in \mathit{Factor}(w) \text{ and } a \in \Sigma\}. \\ E_{R\to L} &= \{(x, a, ax) \mid x, ax \in \mathit{Factor}(w) \text{ and } a \in \Sigma\}. \end{aligned}$$

Definition 6 *$STree(w)$ with $DAWG(w^{\mathrm{rev}})$ is the bidirectional dag $(V, E_{L\to R}, E_{R\to L})$ such that*

$$\begin{aligned} V &= \{\overset{w}{\overrightarrow{x}} \mid x \in \mathit{Factor}(w)\}, \\ E_{L\to R} &= \{(\overset{w}{\overrightarrow{x}}, a\beta, \overset{w}{\overrightarrow{xa}}) \mid x, xa \in \mathit{Factor}(w),\ a \in \Sigma,\ \beta \in \Sigma^*,\ \overset{w}{\overrightarrow{xa}} = xa\beta, \text{ and } \overset{w}{\overrightarrow{x}} \neq \overset{w}{\overrightarrow{xa}}\}, \\ E_{R\to L} &= \{(\overset{w}{\overrightarrow{x}}, a, \overset{w}{\overrightarrow{ax}}) \mid x, ax \in \mathit{Factor}(w) \text{ and } a \in \Sigma\}. \end{aligned}$$

Definition 7 *$SCDAWG(w)$ is the bidirectional dag $(V, E_{L\to R}, E_{R\to L})$ such that*

$$\begin{aligned} V &= \{\overset{w}{\overleftrightarrow{x}} \mid x \in \mathit{Factor}(w)\}, \\ E_{L\to R} &= \{(\overset{w}{\overleftarrow{x}}, a\beta, \overset{w}{\overleftarrow{xa}}) \mid x, xa \in \mathit{Factor}(w),\ a \in \Sigma,\ \beta \in \Sigma^*,\ \overset{w}{\overrightarrow{xa}} = xa\beta, \text{ and } \overset{w}{\overrightarrow{x}} \neq \overset{w}{\overrightarrow{xa}}\}, \\ E_{R\to L} &= \{(\overset{w}{\overleftarrow{x}}, \gamma a, \overset{w}{\overleftarrow{ax}}) \mid x, ax \in \mathit{Factor}(w),\ a \in \Sigma,\ \gamma \in \Sigma^*,\ \overset{w}{\overleftarrow{ax}} = \gamma ax, \text{ and } \overset{w}{\overleftarrow{x}} \neq \overset{w}{\overleftarrow{ax}}\}. \end{aligned}$$

Figure 2. The definitions of the bidirectional indexing structures $STrie(w)$ with $STrie(w^{\mathrm{rev}})$, $STree(w)$ with $DAWG(w^{\mathrm{rev}})$, and $SCDAWG(w)$.

are the set

$$\left\{ (\overset{w}{\overrightarrow{ax}}, \overset{w}{\overrightarrow{x}}) \;\middle|\; \begin{array}{l} x, ax \in \mathit{Factor}(w), a \in \Sigma, \text{ and} \\ \overset{w}{\overrightarrow{ax}} = a \cdot \overset{w}{\overrightarrow{x}} \end{array} \right\}.$$

The reversal of the suffix links are *reversed suffix link* defined as

$$\left\{ (\overset{w}{\overrightarrow{x}}, a, \overset{w}{\overrightarrow{ax}}) \;\middle|\; \begin{array}{l} x, ax \in \mathit{Factor}(w), a \in \Sigma, \text{ and} \\ \overset{w}{\overrightarrow{ax}} = a \cdot \overset{w}{\overrightarrow{x}} \end{array} \right\}.$$

In Fig. 3 we illustrate $STrie(w)$ with $STrie(w^{\mathrm{rev}})$

and $STree(w)$ with $DAWG(w^{rev})$, where $w =$ baggage.

By the duality, we omit the definition of the bidirectional index structure $DAWG(w)$ with $STree(w^{rev})$.

In Definition 7 of Fig. 2, we show the definition of the symmetric CDAWG (SCDAWG) of a string w, denoted by $SCDAWG(w)$, originally defined by Blumer et al. [2]. The edges $E_{R\to L}$ are called the sext links of $CDAWG(w)$, as well. The suffix links of $CDAWG(w)$ are the set

$$\left\{ (\overset{w}{\overrightarrow{ax}}, \overset{w}{\overleftarrow{x}}) \;\middle|\; \begin{array}{l} x, ax \in Factor(w), a \in \Sigma, \\ \text{and} \overset{w}{\overleftarrow{x}} \neq \overset{w}{\overleftarrow{ax}} \end{array} \right\},$$

whereas the reversed suffix link of $CDAWG(w)$ are the set

$$\left\{ (\overset{w}{\overleftarrow{x}}, \gamma a, \overset{w}{\overrightarrow{ax}}) \;\middle|\; \begin{array}{l} x, ax \in Factor(w), a \in \Sigma, \\ \gamma \in \Sigma^*, \overset{w}{\overleftarrow{ax}} = \gamma ax, \\ \overset{w}{\overleftarrow{x}} \neq \overset{w}{\overrightarrow{ax}}, \text{ and } \overset{w}{\overrightarrow{ax}} = a \cdot \overset{w}{\overleftarrow{x}} \end{array} \right\}.$$

We illustrate $DAWG(w)$ with $STree(w^{rev})$, and $SCDAWG(w)$ in Fig. 4, where $w =$ baggage.

Another symmetric bidirectional index structure, called *affix tree*, was introduced by Stoye [12]. $ATree(w)$ and $ATree(w^{rev})$ for $w =$ baggage are shown in Fig. 5 without a formal definition for comparison. Intuitively, the set of the nodes in $SCDAWG(w)$ is the *intersection* of those in $STree(w)$ and $STree(w^{rev})$, while the set of the nodes in $ATree(w)$ is the *union* of them.

3 On-Line Construction of $STree(w)$ with $DAWG(w^{rev})$

In this section, we give an algorithm that simultaneously constructs $STree(w)$ with $DAWG(w^{rev})$ for a string $w \in \Sigma^*$, on-line and in linear time with respect to $|w|$.

3.1 Definition

For any string $w \in \Sigma^*$, let $STree'(w)$ denote the tree obtained by eliminating the non-branching internal nodes from $STree(w)$. The Ukkonen suffix tree construction algorithm builds $STree'(w)$ rather than $STree(w)$ in on-line manner for a string w. Despite the difference between the two trees, we can use the algorithm because we know that $STree'(w\$) = STree(w\$)$ when \$ does not occur in w. Our algorithm constructs $STree'(w)$ in the same fashion as the Ukkonen algorithm, and therefore the $DAWG(w^{rev})$ being constructed at the same time is incomplete in the sense that it lacks the nodes corresponding to the non-branching internal nodes of $STree(w)$ and the sext links from/to them. However, the finally obtained structure for input $w\$$ is exactly the same as $STree(w\$)$ with $DAWG(\$w^{rev})$.

Let us denote by "$STree'(w)$ with sext links" this incomplete version of "$STree(w)$ with $DAWG(w^{rev})$". Before describing our algorithm, we have to formally define $STree'(w)$, and "$STree(w)$ with sext links", in order to clarify what our algorithm constructs.

For a string $w \in \Sigma^*$, let

$$L_w = \left\{ (x, xa) \;\middle|\; \begin{array}{l} x \in Factor(w), \text{and } a \in \Sigma \\ \text{is the unique symbol} \\ \text{such that } xa \in Factor(w). \end{array} \right\},$$

and let $\equiv'^{L}_{w}$ be the equivalence closure of L_w, i.e., the smallest superset of L_w that is symmetric, reflexive, and transitive. It can be readily shown that $\equiv^{L}_{w}$ is a refinement of $\equiv'^{L}_{w}$, namely, every equivalence class in $\equiv'^{L}_{w}$ is a union of one or more equivalence classes in $\equiv^{L}_{w}$. For a string $x \in Factor(w)$, denote by $\overset{w}{\Longrightarrow}{x}$ the longest string in the equivalence class to which x belongs under the equivalence relation $\equiv'^{L}_{w}$. We remark that $\overset{w}{\overrightarrow{x}}$ is a prefix of $\overset{w}{\Longrightarrow}{x}$, and that $\overset{w}{\Longrightarrow}{x}$ corresponds to the 'locus' of a string $x \in Factor(w)$ in $STree'(w)$, not in $STree(w)$.

Now we are ready to give formal definitions.

Definition 8 *$STree'(w)$ is the tree (V, E) obtained by replacing the $\overset{w}{\overrightarrow{(\cdot)}}$ operation with the $\overset{w}{\Longrightarrow}{(\cdot)}$ operation in Definition 2.*

Definition 9 *$STree'(w)$ with sext links is the bidirectional dag $(V, E_{L\to R}, E_{R\to L})$ obtained by replacing the $\overset{w}{\overrightarrow{(\cdot)}}$ operation with the $\overset{w}{\Longrightarrow}{(\cdot)}$ operation in Definition 6.*

Since we can prove that $\overset{w\$}{\Longrightarrow}{x} = \overset{w\$}{\overrightarrow{x}}$, these structures are identical to those defined in Definition 2 and Definition 6, respectively, for input string $w\$$.

3.2 Main Idea of the Algorithm

As for $STree'(w)$ for a string $w \in \Sigma^*$, our algorithm creates it in entirely the same way as the Ukkonen algorithm. Every time a new node is created during the construction of $STree'(w)$, the sext

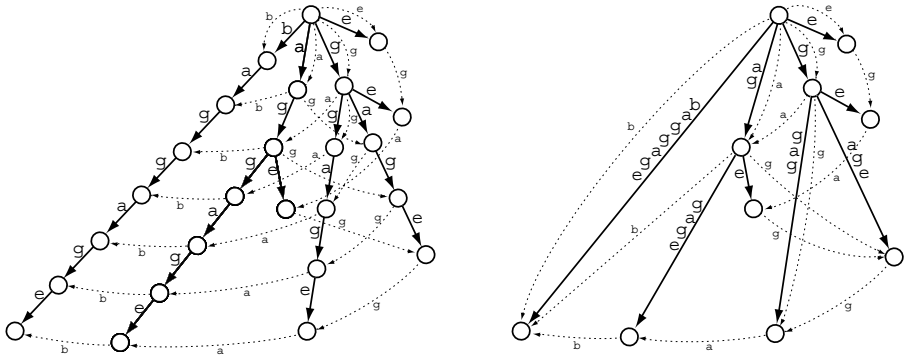

Figure 3. $STrie(w)$ **with** $STrie(w^{\rm rev})$ **at the left, and** $STree(w)$ **with** $DAWG(w^{\rm rev})$ **at the right, where** $w = \texttt{baggage}$**. The thick solid lines represent the edges of** $STrie(w)$ **and** $STree(w)$**, while the thin break lines do the ones of** $STrie(w^{\rm rev})$ **and** $DAWG(w^{\rm rev})$**. Since the string** baggage **ends with a unique character** e**, the endmarker** \$ **is omitted.**

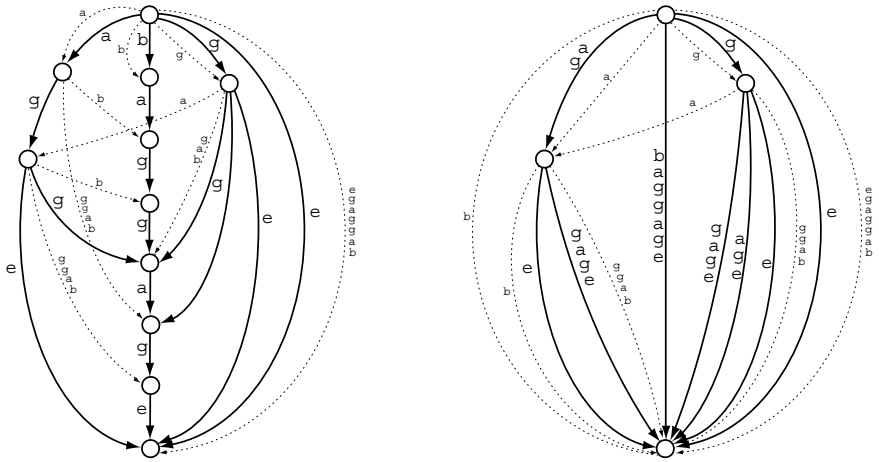

Figure 4. $DAWG(w)$ **with** $STree(w^{\rm rev})$ **at the left, and** $SCDAWG(w)$ **at the right, for string** $w = \texttt{baggage}$**.**

links of the new node, which correspond to certain edges of $DAWG(w^{\rm rev})$, are computed. The Ukkonen algorithm creates no leaf node for the use of so-called "∞-trick" that enables his algorithm to achieve an $O(|w|)$-time construction of $STree'(w)$, and an edge directed to a "transparent" leaf node is called an *open* edge. However, we modify it so as to create every leaf node not only because

(i) we need a leaf node to define its sext links, but also

(ii) the sext link of a leaf node is to be a clue to define the sext links of a node to be created just above the leaf node.

First of all, one may wonder that if creating leaf nodes, the time complexity of the construction of $STree'(w)$ can be quadratic due to a series of updating the open edges. However, recall the fact that label α of an edge of $STree'(w)$ is usually implemented with a pair of integers (i, j) such that $\alpha = w[i : j]$. Furthermore, note that the second value of the label of any open edge in $STree'(w[1 : h])$ is h for $1 \leq h \leq n$. Therefore, if we implement the second value with a global variable, we can update all the open edges in constant time with increment of the variable h.

Let us pay our attention back to the two reasons (i) and (ii). We have an obvious proposition about

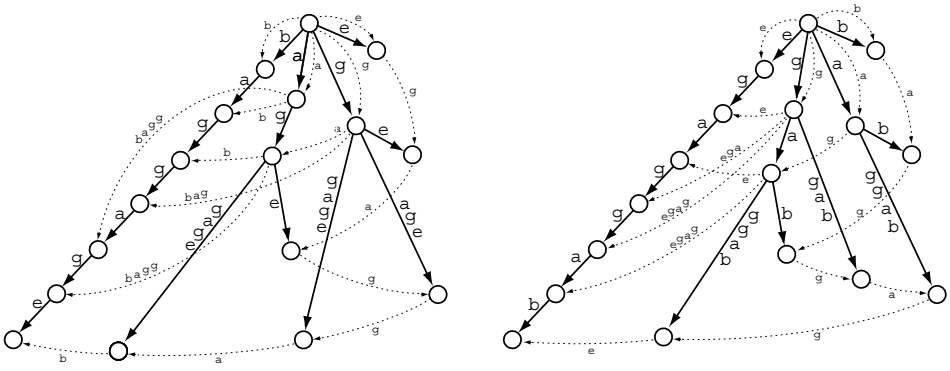

Figure 5. $ATree(w)$ **at the left and** $ATree(w^{\mathrm{rev}})$ **at the right, where** $w =$ `baggage`**.**

(i).

Proposition 1 *Suppose that in $STree'(w)$ the reversed suffix link of a leaf node x, which is labeled a, points to a node y. Then node y is also a leaf node in $STree'(w)$.*

Proof. From the definition the reversed suffix link of node x is a triple $(\overrightarrow{\overrightarrow{x}}^{w}, a, \overrightarrow{\overrightarrow{ax}}^{w})$ such that $\overrightarrow{\overrightarrow{ax}}^{w} = a \cdot \overrightarrow{\overrightarrow{x}}^{w}$. String x is a suffix of w because x is represented by a leaf in $STree'(w)$. Hence $\overrightarrow{\overrightarrow{x}}^{w} = x$. Consequently, $\overrightarrow{\overrightarrow{ax}}^{w} = a \cdot \overrightarrow{\overrightarrow{x}}^{w} = ax = y$. This means that y is also a suffix of w and is represented by a leaf node in $STree'(w)$. □

The above proposition tells us that, in a suffix tree, the reversed suffix link of the newest leaf node points to the last created leaf node. Conversely, the suffix link of the last created leaf node is pointing to the leaf node which will be created next.

In the sequel, we shall clarify what the reason (ii) implies.

On the construction of "$STree'(w)$ with sext links", we use a two dimensional table *sext*. The description "$sext[x, a] = y$" means "the sext link of node x labeled with a points to node y." Similarly, we use tables *suf* and *rsuf* which correspond to the suffix link and the reversed suffix link, respectively.

3.3 How to Maintain Sext Links

Here, we explain how the sext links of a new node are computed during the Ukkonen-type construction of $STree'(w)$. See Fig. 6 that shows each phase of the construction of $STree'(\#\texttt{abab\$})$. The starred point in Fig. 6 is called the *active point*. For a string $w \in \Sigma^*$, at the beginning of each phase $w[1:i]$ ($i = 0, 1, \ldots, |w| - 1$), the active point stays at which the algorithm should start to update $STree'(w[1:i])$ to $STree'(w[1:i+1])$. Let act_i denote the active point in phase $w[1:i]$. In phase $w[1:i+1]$, act_{i+1} moves until it can stop with spelling out $w[i+1]$.

If it is possible for act_{i+1} to move ahead from the current location while spelling out $w[i+1]$ (say case (a)), it moves and stops there, and then becomes act_{i+2}. Notice that no new node is created in case (a), as seen in phase $\#\texttt{aba}$ and phase $\#\texttt{abab}$ in Fig. 6. Otherwise (say case (b)), a new edge labeled with $w[i+1]$ has to be created from where act_{i+1} currently stays. Case (b) is divided into two sub-cases:

- act_{i+1} is on a node u (case (b_1)).
- act_{i+1} is on an edge (case (b_2)).

In case (b_1), the algorithm just creates a new edge labeled by $w[i+1]$ with a new leaf node v (see Fig. 7, left). Only v is the newly created node in case (b_1). Concrete examples can be seen in phases $\#$, $\#\texttt{a}$, $\#\texttt{ab}$, and the third step of phase $\#\texttt{abab\$}$ in Fig. 6. As for case (b_2), the algorithm needs to create a new node u where act_{i+1} stays now, in the middle of the edge, to insert a new edge labeled with $w[i+1]$ from there (see Fig. 7, right). Concrete examples of case (b_2) can bee seen at the first and second steps in phase $\#\texttt{abab\$}$ in Fig. 6. After having making node u, it creates a new edge together with a new leaf node v. These nodes u and v are all the nodes newly created in case (b_2).

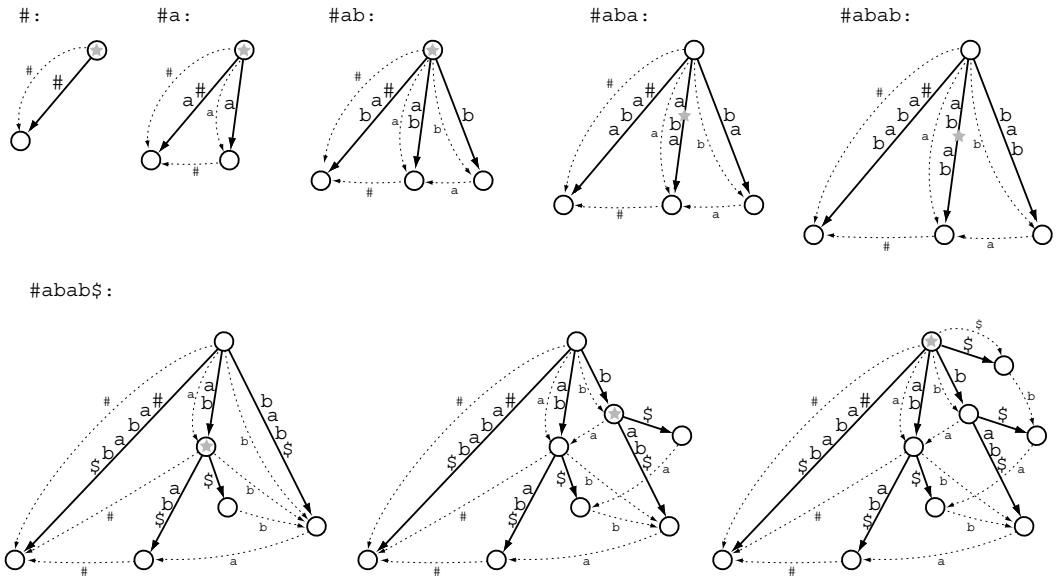

Figure 6. The on-line construction of $STree'(\#\texttt{abab\$})$ with the sext links represented by the broken arrows. At the third step of phase $\#\texttt{abab\$}$, the sext links form $DAWG(\$\texttt{baba}\#)$.

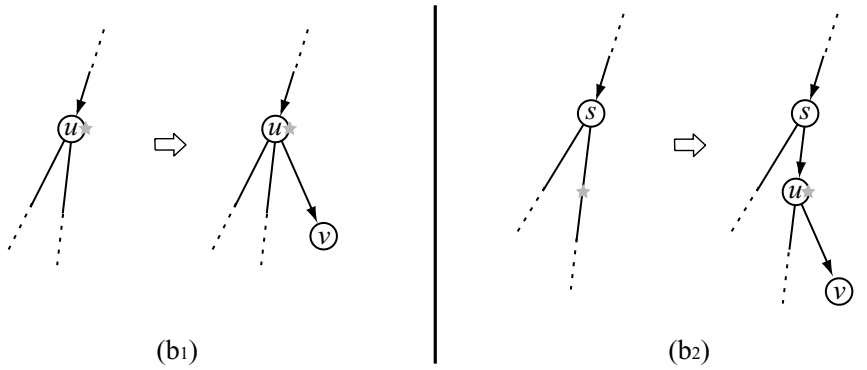

Figure 7. The two cases of the position of the active point, which is denoted by a gray star. Since the active point is on a node u in case (b_1) displayed on the left, only leaf node v is newly created. On the other hand, in case (b_2) on the right, internal node u is also created where the active point is at present, in the middle of an edge.

3.3.1 Sext Link of a Leaf Node

In both cases (b_1) and (b_2), it follows from Proposition 1 that the reversed suffix link of a new leaf node v points to the last created leaf node v'. Suppose v is the jth created leaf node and v' is $(j-1)$th one during the construction of $STree'(w)$, where $2 \leq j \leq |w|$. Then the reversed suffix link of node v pointing to v' is labeled by $w[j-1]$, in formula, $rsuf[v, w[j-1]] = v'$. We have the following proposition which concerns with the sext link of v.

Proposition 2 *Suppose that v and v' are jth and $(j-1)th$ created leaf nodes of $STree'(w)$, respectively, where $1 \leq j \leq |w|$. Then $sext[v, w[j-1]] = v'$ is the sole sext link of leaf node v.*

Proof. Since v is a leaf node, v is a factor which has occurred only once in w, as a suffix. Because

v is the jth suffix, it is preceded by $w[j-1]$ and $w[j-1] \cdot v = v'$. Therefore, for any $c \in \Sigma$ such that $c \neq w[j-1]$, the string cv is not in $Factor(w)$. □

For example, leaf node b is created in phase #ab of Fig. 6, and it is the *third* one. Therefore, $rsuf[\mathtt{b}, \mathtt{a}] = sext[\mathtt{b}, \mathtt{a}] = \mathtt{ab}$, where a is the *second* character in string #abab$.

3.3.2 Sext Links of an Internal Node

Since the leaf node v is the only node newly created in case (b_1), the algorithm then has only to do the above maintenance for node v. Meanwhile, because the node u is also newly created in case (b_2), we have to determine the sext links of u. Assume that in phase $w[1:i]$ the internal node u is created in the middle of an edge between node s and node r. It then results in that u has two children, r and v. If there exists a node u' such that $suf[u'] = u$, then let a be the character such that $rsuf[u, a] = u'$. Suppose there is a node r' such that $sext[r, b] = r'$ with $b \neq a$. Then $sext[u, b]$ is set to point to r' as well, since $r' = \overset{w[1:i]}{\Longrightarrow} bu$ in this case (remember the definition of sext links). For instance, at the first step of phase #abab$ in Fig. 6, $u = \mathtt{ab}$, $r = \mathtt{abab\$}$ and $r' = sext[\mathtt{abab\$}, \#] = \#\mathtt{abab\$}$. Since $rsuf[\mathtt{ab}, \#]$ is undefined, we define $sext[\mathtt{ab}, \#] = \#\mathtt{abab\$}$. If $b = a$, then $sext[u, b]$ stays pointing to node u', because obviously $u' = \overset{w[1:i]}{\Longrightarrow} bu = \overset{w[1:i]}{\Longrightarrow} au$. For example, at the second step of phase #abab$ in Fig. 6, $sext[\mathtt{bab\$}, \mathtt{a}] = \mathtt{abab\$}$. However, since $rsuf[\mathtt{b}, \mathtt{a}] = \mathtt{ab}$, $sext[\mathtt{b}, \mathtt{a}] = \mathtt{ab}$. As previously remarked in the reason (ii) in Section 3.2, we also refers to the sext link of leaf node v in order to determine the sext links of node u, in the same way as mentioned above about the sext links of r. Formally, we have the following lemma.

Lemma 1 *When an internal node u is newly created in phase $w[1:i]$ during the construction of $STree'(w)$ with sext links, let r be the existing child node of u and v be the new leaf node which is also a child of u. Then, $sext[u, c]$ is created for each character c such that either $sext[r, c]$ or $sext[v, c]$ was present at the beginning of the phase.*

Proof. It follows from the definition that a node x has a sext link labeled by a character c if and only if an occurrence of the string x is preceded by c. Note that the string u is a suffix of the string $w[1:i]$, and that each of the occurrences of u within $w[1:i-1]$ is followed by the string α such that $u\alpha = r$. Therefore, if there is an occurrence of u within $w[1:i-1]$ which is preceded by c, then the node r has a sext link labeled by c. On the other hand, if c is the preceding character of the occurrence of u within $w[1:i]$ that ends at the last character of $w[1:i]$, then the node v has a sext link labeled by c. □

On the other hand, if the active point arrives at a node when case (a) is applied, a new sext link of the node is created. Suppose that, just after a leaf node v had been created, the active point stopped on a node in phase $w[1:i]$ during the construction of $STree'(w)$, where $1 \leq i \leq |w|$. In addition, assume that v is the jth created leaf node, where $1 \leq j \leq |w|$. That is to say, $v = w[j:i]$. Notice that $j \leq i$. After that, if the active point stops on a node p with case (a) in the next phase, phase $w[1:i+1]$, then a sext link of node p which is labeled $w[j]$ is created and set to point to node v, where v now represents $w[j:i+1]$. Let us clarify the reason for the above. Let u and u' be the parent nodes of v and p, respectively. Notice that then $u \cdot w[i:i+1] = v = w[j:i+1]$. Furthermore, $u' \cdot w[i+1:i+1] = u' \cdot w[i+1] = w[j+1:i+1]$ since $suf[u] = u'$. Namely, node v currently represents $w[j:i+1]$ and node p corresponds to $w[j+1:i+1]$. That is why $sext[p, w[j]] = v$. If the active point again stops on a node until the algorithm faces case (b), the sext link of the node whose label is $w[j]$ is created and set to point to the leaf node v as well. A concrete example is shown in Fig. 8.

3.3.3 Sext Links Pointing to a New Node

The only thing we have not accounted for yet is to change the sext links that point to the newly created nodes u and v. Let us first mention the case of v, a new leaf node. The following remark about a new leaf node v is common to case (b_1) and case (b_2). Whenever a character $w[i]$ appears in string $w[1:i]$ for the first time, a new edge labeled with $w[i]$ is created from the root node, and v is associated with $w[i]$. Then, $sext[\varepsilon, w[i]] = v$, because the root node corresponds to the empty string ε. This can be seen in phases #, #a, #ab, and #abab$ in Fig. 6. If the character $w[i]$ has already appeared in string $w[1:i-1]$, then leaf node v should be pointed to by the leaf node which will be created next.

We now treat how to decide what sext link of $STree'(w[1:i])$ should be modified so as to point to a newly created internal node u, in case (b_2). Recall that node u has two children r and v. Let

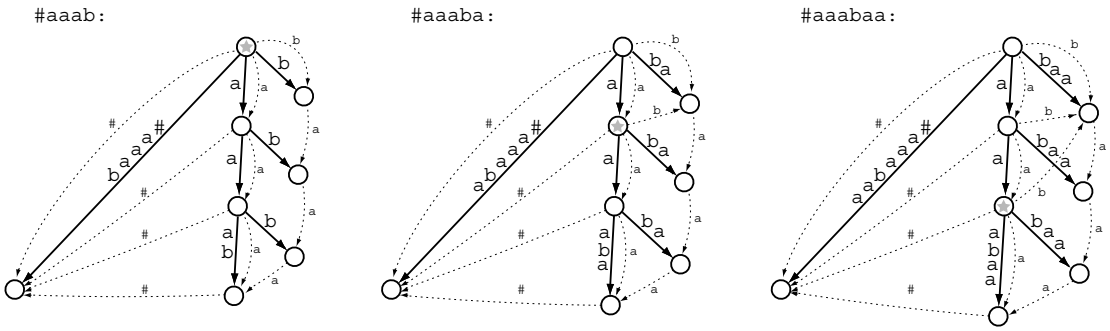

Figure 8. $STree'(\#\texttt{aaab})$ **with the sext links is shown on the left. Node** `b` **is the last created leaf node in that phase. Scanning a new character** `a`**, the active point moves to node** `a`**, as seen in the center figure** $STree'(\#\texttt{aaaba})$**. Then,** $sext[\texttt{a},\texttt{b}]$ **is set to point to the last created leaf node** `ba`**. Also in the right figure representing** $STree'(\#\texttt{aaabaa})$**,** $sext[\texttt{aa},\texttt{b}] = \texttt{baa}$**, because the active point has arrived at node** `aa`**.**

us suppose that node r is pointed to by a c-labeled sext link of a node p in $STree'(w[1:j])$ where $j = i-1$, that is, $sext[p,c] = r$. In other words, $\overset{w[1:j]}{\Longrightarrow} cp = r$. If $|u| > |p|$, then the sext link of p is modified so as to point to u ($sext[p,c] = u$), because $\overset{w[1:i]}{\Longrightarrow} cp = u$. A concrete example can be seen between phase $\#\texttt{abab}$ and phase $\#\texttt{abab\$}$ in Fig. 6. $sext[\varepsilon,\texttt{a}] = \texttt{abab}$ in phase $\#\texttt{abab}$ is modified as $sext[\varepsilon,\texttt{a}] = \texttt{ab}$ at the first step of phase $\#\texttt{abab\$}$, where node `ab` is the internal node newly created in phase $\#\texttt{abab\$}$. In another case (if $|u| \leq |p|$), the sext link of node p remains pointing to node r, since $\overset{w[1:i]}{\Longrightarrow} cp = r$ in this case. Similar discussion holds for the sext links pointing to node v, another child of node u.

3.4 Correctness and Complexity of the Algorithm

The algorithm is summarized as Fig. 9 and Fig. 10. If we compute the sext links of the nodes in "$STree'(w)$ with sext links" according to the algorithm, we have the following:

Theorem 1 *For any string $w \in \Sigma^*$, $STree'(w)$ with sext links can be constructed on-line and in linear time and space with respect to $|w|$.*

Proof. Since it has been proven in [15] that $STree'(w)$ can be obtained on-line and in $O(|w|)$ time, all we have to clarify are the correctness and complexity of the construction of sext links. The data structure we newly add to the Ukkonen algorithm are the table *sext* and *rsuf*. It is clear that they require $O(|\Sigma|\cdot|w|)$ space. Therefore, if Σ is a fixed alphabet, the space complexity of our algorithm is linear.

We have assumed that a string w ends with a unique end-marker \$. After \$ is scanned, a new edge labeled with \$ is absolutely created from the root node, and the corresponding new leaf node is also created. After that, the sext link of the root node, which is labeled \$, is set to point to the new leaf node. Then, the chain formed by the sext links of all the leaf nodes in $STree'(w)$ exactly spells w^{rev}, i.e., the path of $DAWG(w^{rev})$ which corresponds to string w^{rev} is then completed. This guarantees that the paths of $DAWG(w^{rev})$ corresponding to the suffixes of w^{rev} are also created as the sext links of the internal nodes of $STree'(w)$. This algorithm constructs $DAWG(w^{rev})$ on-line, because new sext links are computed each time a new node is created.

From here on, we establish the sext links can be computed in linear time with respect to $|w|$. It is obvious that to decide the sext link of any new leaf node takes only constant time. When we determine the sext links of a newly created internal node, we copy the sext links of the two children of the new node. It takes $O(|\Sigma|)$ time, since each of the two children has at most $|\Sigma|$ sext links. Therefore, if Σ is a fixed alphabet, it takes constant time. The matter is the change of sext links due to a new-created internal node. Suppose that, in phase $w[1:i]$, act_i stays somewhere depth m in $STree'(w[1:i])$. At the beginning of phase $w[1:i+1]$, the algo-

Algorithm Construction of $STree'(text\$)$ with sext links in alphabet $\Sigma = \{text[-1], \ldots, text[-m]\}$, and \$ is the end marker not appearing elsewhere in $text$.

```
 1 create nodes root and bottom;
 2 for j := 1 to m do create a new edge (bottom, (-j, -j), root);
 3 suf[root] := bottom;
 4 length(root) := 0;    length(bottom) := -1;
 5 (s, k) := (root, 1); i := 0;
 6 lastleaf := nil;    n := 0;    /* lastleaf is the last (n-th) created leaf node */
 7 repeat
 8     i := i + 1;
 9     (s, k, lastleaf, n) := update(s, (k, i), lastleaf, n);
10 until text[i] = $;

function update(s, (k, p), lastleaf, n):
/* (s, (k, p - 1)) is the canonical reference pair for the active point. */
 1 oldr := nil; s' := nil;
 2 while not check_end_point(s, (k, p - 1), text[p]) do
 3     if k ≤ p - 1 then    /* implicit */
 4         s' := extension(s, (k, p - 1));
 5         r := split_edge(s, (k, p - 1));
 6     else    /* explicit */
 7         r := s;
 8     create a new leaf node v and a new edge (r, (p, e), v);
 9     /* e is the global variable representing the scanned length of the input string. */
10     let length(v) be e - n;
11     if oldr ≠ nil then set_suffix_link(oldr, r);
12     if lastleaf ≠ nil then set_suffix_link(lastleaf, v);
13     if r ≠ s then    /* maintenance of sext links */
14         c := text[n];
15         if rsuf[r, c] = nil then sext[r, c] := sext[v, c];
16         for each character a such that sext[s', a] ≠ nil do
17             if rsuf[r, a] = nil then sext[r, a] := sext[s', a];
18         for each sext link sext[x, a] = s' do /* modify sext links pointing to s' */
19             if length(r) > length(x) then sext[x, a] := r;
20     oldr := r;    lastleaf := v;    n := n + 1;
21     (s, k) := canonize(suf[s], (k, p - 1));
22 if oldr ≠ nil then set_suffix_link(oldr, s);
23 (s, k) := canonize(s, (k, p));
24 if k > p then sext[s, text[n]] := lastleaf;
25 return (s, k, lastleaf, n);
```

Figure 9. Main routine and function $update$.

rithm begins to seek for the location where the active point can stop. Then, at most m sext links are changed until the active point stops. This implies that the overall complexity of the change of sext links due to new internal nodes takes $O(|w|)$ time. □

4 On-Line Construction of SCDAWG

In this section, we propose how to construct SCDAWG for a string w, on-line in $O(|w|)$ time. Define $CDAWG'(w)$ and $SCDAWG'(w)$ in a similar way to the definition of $STree'(w)$. Our on-line algorithm builds $CDAWG'(w)$ in the same way as in [9], and builds certain edges of $CDAWG(w^{rev})$

procedure *set_suffix_link*(s, t):
1 let c be the first character of the string represented by s;
2 *suf*$[s] := t$; *rsuf*$[t, c] := s$; $sext[t, c] := s$;

function $check\_end\_point(s, (k, p), c)$: **boolean**;
1 **if** $k \leq p$ **then** /* *implicit* */
2 let $(s, (k', p'), s')$ be the $text[k]$-edge from s;
3 **return** $(c = text[k' + p - k + 1])$;
4 **else** /* *explicit* */
5 **return** (there is a c-edge from s);

function $extension(s, (k, p))$: node;
/* $(s, (k, p))$ *is a canonical reference pair.* */
1 **if** $k > p$ **then return** s; /* *explicit* */
2 find the $text[k]$-edge $(s, (k', p'), s')$ from s; **return** s'; /* *implicit* */

function $canonize(s, (k, p))$: pair of **integers**; /* *borrowed from the Ukkonen algorithm.* */
1 **if** $k > p$ **then return** (s, k); /* *explicit* */
2 find the $text[k]$-edge $(s, (k', p'), s')$ from s;
3 **while** $p' - k' \leq p - k$ **do**
4 $k := k + p' - k' + 1$; $s := s'$;
5 **if** $k \leq p$ **then** find the $text[k]$-edge $(s, (k', p'), s')$ from s;
6 **return** (s, k);

function $split\_edge(s, (k, p))$: node;
1 let $(s, (k', p'), s')$ be the $text[k]$-edge from s;
2 replace this edge by edges $(s, (k', k' + p - k), r)$ and $(r, (k' + p - k + 1, p'), s')$, where r is a new node;
3 $length(r) := length(s) + (p - k + 1)$;
4 **return** r;

Figure 10. Other functions.

as the sext links of the nodes of $CDAWG'(w)$.

We stress that the algorithm of [9] is based on the Ukkonen suffix tree construction algorithm. This implies, if we add the functions "*redirect*" and "*separate_node*" in [9] to the pseudo-code of the algorithm in Section 3, we obtain $CDAWG'(w)$. The matter is how to build the edges of $CDAWG(w^{\mathrm{rev}})$, the sext links of $CDAWG(w)$, of course. However, we fortunately have the fact that CDAWGs can have "the same amount of information" as suffix trees. The loss of information comes from the property that CDAWGs have a node having two or more incoming edges, which correspond to two or more nodes connected by suffix links in suffix trees. Namely, the lost information is strings obtained by concatenating labels of some suffix links. One hint has been given in [7] as an exercise. Furthermore, the CDAWG construction algorithm in [9] is capable of storing the "lost" information as integers in nodes. Notice that if we can treat CDAWGs like suffix trees, it means we can obtain the sext links of CDAWGs.

In the following subsections, we show how the algorithm of Section 3 should be changed when constructing CDAWGs, by using examples. If again turning our attention to the pseudo-code, the 8th line of *update* function is changed to as "create a new edge $(r, (p, e), final)$;" and labels of reversed suffix links and sext links can be of strings, not a character.

4.1 Sext Link Corresponding to a Newly Created Edge

A sequence of snapshots on the on-line construction of $SCDAWG'(\#\texttt{abab\$})$ is shown in Fig. 11. Since character `a` has appeared in string $\#\texttt{a}$, the edge labeled with `a` is created and directed

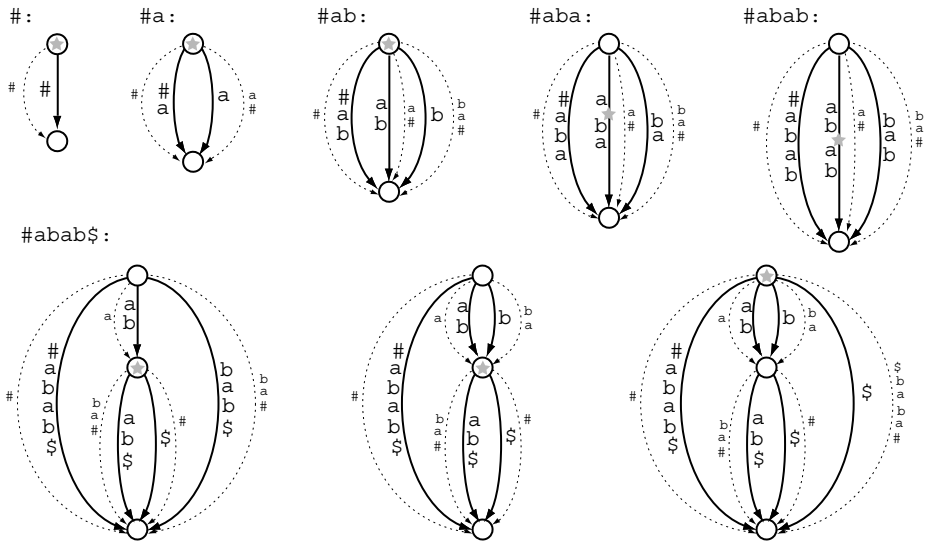

Figure 11. The on-line construction of $SCDAWG'(w)$, where $w = \#\texttt{abab}\$$. The solid arrows represent the edges of $CDAWG'(w)$, whereas the broken arrows represent the sext links of the nodes of $CDAWG'(w)$, that are equivalent to the edges of $CDAWG(w^{rev})$.

to the final node in phase $\#\texttt{a}$. After that, the sext link of the initial node labeled with $\texttt{a}\#$ is set to point to the final node. Comparing it with the corresponding phase in Fig. 6, one can see that character $\#$ in the label $\texttt{a}\#$ of the CDAWG corresponds to the label $\#$ of the sext link from the leaf node $\texttt{a}$ to node $\#\texttt{a}$ of the suffix tree in the phase $\#\texttt{a}$. In general, in phase $w[1:i]$ of the construction of $CDAWG'(w)$, the representative of the final node is $w[1:i]$. Assume that an edge is then created from a node u and it is the jth edge entering to the final node, where $1 \leq j \leq i$. Then, the jth edge is associated with $w[j:i]$. There then exists a "gap" $w[1:j-1]$ between the representative $w[1:i]$ and $w[j:i]$. Notice that this "gap" corresponds to the reversal of the concatenation of the labels of the sext links between leaf node $w[j:i]$ and leaf node $w[1:i]$ in $STree(w[1:i])$. On the grounds of this gap $w[1:j-1]$, a new sext link of node u is set to point to the final node with label $(w[1:j])^{rev}$.

4.2 Change of Sext Links

See phases $\#\texttt{abab}$ and $\#\texttt{abab}\$$ in Fig. 11. The active point stays on the middle of the edge labeled $\texttt{abab}$ in phase $\#\texttt{abab}$, and the edge is split into two edges due to the creation of the new edge labeled $\$$. Notice that the sext link labeled with $\texttt{a}\#$ is also cut into two. One labeled with $\texttt{a}$ is set to point to the new node $\texttt{ab}$, and the other labeled $\#$ is set from node $\texttt{ab}$. It is because $\overset{w[1:6]}{\Longrightarrow}\varepsilon a = ab$ and $\overset{w[1:6]}{\Longrightarrow}\#ab = \#abab\$$ in this time, where $w[1:6] = \#abab\$$.

What if a sext link, whose label is of length more than 1, is cut? See Fig. 12 that displays $CDAWG'(\#\texttt{abb})$ and $CDAWG'(\#\texttt{abba})$. There is a sext link of the initial node pointing to the final node, which is labeled with $\texttt{ba}\#$ in $CDAWG'(\#\texttt{abb})$. At the beginning of the conversion to $CDAWG'(\#\texttt{abba})$, a new node $\texttt{b}$ is created where the active point currently stays. Then, the sext link labeled $\texttt{ba}\#$ is cut and its former part is set to point to the new node $\texttt{b}$, labeled with $\texttt{b}$. In general, if a new node is created in the middle of an edge, the sext link corresponding to the edge is cut into two, and its former part is labeled with the *single* initial character of the label of the cut sext link. It does not depend on the length of the label of the sext link to be cut.

To realize the operation above mentioned, we need to associate the sext link labeled $\texttt{ba}\#$ with *string* $\texttt{bb}$ in the final node, where $\texttt{bb}$ is not the representative of the final node. This is because if we associate that just with the representative, like $sext[\varepsilon, \texttt{ba}\#] = \#\texttt{abb}$, we cannot recognize which sext link pointing to the final node should be cut owing to the newly created node (notice there exist other sext links from the initial node to the final

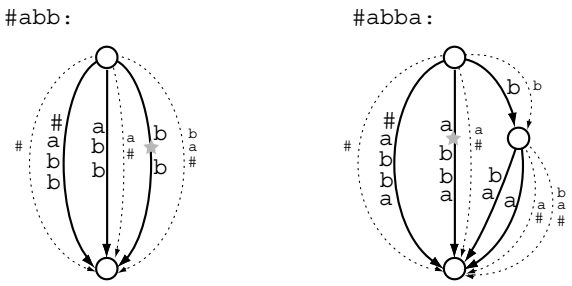

Figure 12. $CDAWG'(\#\texttt{abb})$ **and** $CDAWG'(\#\texttt{abba})$ **with their sext links.**

node). Therefore, we make a sext link point to a string represented in a certain node, not to the representative. For example, on $CDAWG'(\#\texttt{abb})$ in Fig. 12, $sext[\varepsilon, \#] = \#\texttt{abb}$, $sext[\varepsilon, \texttt{a}\#] = \texttt{abb}$, $sext[\varepsilon, \texttt{ba}\#] = \texttt{bb}$.

As seen in phase $\#\texttt{abab\$}$ of Fig. 11, the edge labeled with bab\$ is *merged* into the node ab and its label is modified to b. According to it, $sext[\varepsilon, \texttt{ba}\#] = \texttt{bab\$}$ becomes $sext[\varepsilon, \texttt{ba}] = \texttt{b}$. The character a at the tail of label ba of the sext link corresponds to the label of the sext link from node b to ab in $STree(\#\texttt{abab\$})$ in Fig. 6.

Fig. 13 displays a node *separation* that can happen during the construction of CDAWGs. In Fig. 13, as the active point arrives at node ab via the edge labeled b which belongs to a non-longest path from the initial node to the node ab, the node is cloned as seen in the $CDAWG'(\#\texttt{ababcb})$. Then, $sext[\varepsilon, \texttt{ba}] = \texttt{b}$ in $CDAWG'(\#\texttt{ababc})$ is cut into two, one of which is $sext[\texttt{b}, \texttt{a}] = \texttt{ab}$ and the other $sext[\varepsilon, \texttt{b}] = \texttt{b}$.

4.3 Implementation of Factors Represented in a Node

As is mentioned above, a sext link in $CDAWG'(w)$ is set to point to a certain factor of w represented in a node. However, if we actually implement all of such strings naively, the space requirement can be quadratic. Therefore, we implement them with integers referring to the positions in the input string w. Suppose that the representative of a node p is α in $CDAWG'(w[1:i])$ for $1 \leq i \leq |w|$. Then, node p has integers j and k $(1 \leq j \leq k \leq i)$ such that $\alpha = w[j\!:\!k]$ where j represents the beginning position of the left most occurrence of α in $w[1\!:\!i]$. In addition to it, each edge entering node p has an integer representing the entrance order to node p. See the left figure in Fig. 13, $CDAWG'(\#\texttt{ababc})$. For example, the edge labeled ab is the first one and the edge labeled b is the second one entering to node ab. Note that, in $CDAWG'(\#\texttt{ababc})$, the edge labeled abc entering to the final node represents two factors ababc and babc, which are the second and the third members of the final node, respectively. Thus the edge labeled abc is associated with the set $\{2, 3\}$. In this way, the edges entering to the final node are associated with the sets $\{1\}$, $\{2, 3\}$, $\{4, 5\}$, $\{6\}$ from left to right. In general, an edge in a node may correspond to more than two strings represented in the node. However, the truth is that such strings always occur sequentially in string w, for any w. Therefore, even if an edge corresponds to more than two strings, we can represent all of them with a pair of integers, the minimum and the maximum elements in the set associated with. As a result of the above discussions, we now have:

Theorem 2 *For any string $w \in \Sigma^*$, SCDAWG for w can be constructed on-line in linear time and space with respect to $|w|$.*

5 Conclusions

First, we gave an on-line linear time algorithm to construct the suffix tree for a string together with the DAWG for the reversal of the string. It builds the suffix tree based on Ukkonen's on-line algorithm [15], and simultaneously builds the DAWG as the sext links of the nodes of the suffix tree.

Blumer et al. [2] gave an off-line linear time algorithm for the construction of the SCDAWG for a string: first builds the DAWG with the suffix links, and then compacts the DAWG and its suffix links to the SCDAWG. Meanwhile, the algorithm we proposed in this paper directly constructs the SCDAWG for a given string, on-line in linear time: builds the CDAWG for the string on-line, with the sext links that compose the CDAWG for the reversal of the string. This enables us to save time and

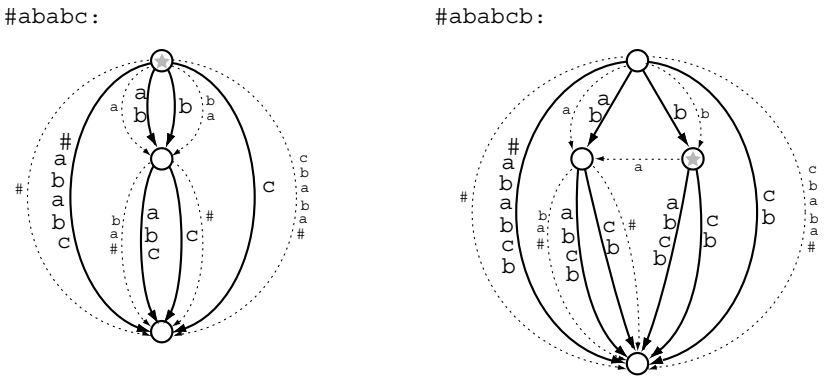

Figure 13. $CDAWG'(\#\texttt{ababc})$ **and** $CDAWG'(\#\texttt{ababcb})$ **with their sext links.**

space at the same time when constructing an SC-DAWG.

References

[1] A. Blumer, J. Blumer, D. Haussler, A. Ehrenfeucht, M. T. Chen, and J. Seiferas. The smallest automaton recognizing the subwords of a text. *Theor. Comput. Sci.*, 40:31–55, 1985.

[2] A. Blumer, J. Blumer, D. Haussler, R. McConnell, and A. Ehrenfeucht. Complete inverted files for efficient text retrieval and analysis. *J. ACM*, 34(3):578–595, 1987. Preliminary version in: STOC'84.

[3] M. T. Chen and J. Seiferas. Efficient and elegant subword tree construction. In A. Apostolico and Z. Galil, editors, *Combinatorial Algorithm on Words*, volume 12 of *NATO Advanced Science Institutes, Series F*, pages 97–107. Springer-Verlag, 1985.

[4] M. Crochemore and W. Rytter. *Text Algorithms*. Oxford University Press, New York, 1994.

[5] M. Crochemore and R. Vérin. On compact directed acyclic word graphs. In *Structures in Logic and Computer Science (LNCS1261)*, pages 192–211. Springer-Verlag, 1997.

[6] R. Giegerich and S. Kurtz. From Ukkonen to McCreight and Weiner: A unifying view of linear-time suffix tree construction. *Algorithmica*, 19(3):331–353, 1997.

[7] D. Gusfield. *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press, New York, 1997.

[8] J. Holub and B. Melichar. Approximate string matching using factor automata. *Theor. Comput. Sci.*, 249:305–311, 2000.

[9] S. Inenaga, H. Hoshino, A. Shinohara, M. Takeda, S. Arikawa, G. Mauri, and G. Pavesi. On-line construction of compact directed acyclic word graphs. In *Proc. 12th Ann. Symp. on Combinatorial Pattern Matching (LNCS2089)*, pages 169–180, July 2001.

[10] M. G. Maaß. Linear bidirectional on-line construction of affix trees. In *Proc. 11th Ann. Symp. on Combinatorial Pattern Matching (LNCS1848)*, pages 320–334. Springer-Verlag, 2000.

[11] E. M. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 23(2):262–272, Apr. 1976.

[12] J. Stoye. Affixbäume. Master's thesis, Universität Bielefeld, May 1995.

[13] J. Stoye. Affix trees. Technical Report 2000–4, Universität Bielefeld, Technische Fakultät, 2000.

[14] M. Takeda, T. Matsumoto, T. Fukuda, and I. Nanri. Discovering characteristic expressions from literary works: A new text analysis method beyond n-gram and KWIC. *Theor. Comput. Sci.*, 2001. (to appear).

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

[16] P. Weiner. Linear pattern matching algorithms. In *Proc. 14th Ann. Symp. on Switching and Automata Theory*, pages 1–11, Oct. 1973.