On-Line Construction of Compact Directed Acyclic Word Graphs

Shunsuke Inenaga [a,b] Hiromasa Hoshino [a] Ayumi Shinohara [a,b]
Masayuki Takeda [a,b] Setsuo Arikawa [a] Giancarlo Mauri [c]
Giulio Pavesi [c]

[a] *Department of Informatics, Kyushu University, Japan*
[b] *PRESTO, Japan Science and Technology Agency (JST)*
[c] *Department of Computer Science, Systems and Communication, University of Milan-Biccoca, Italy*

Abstract

Many different index structures, providing efficient solutions to problems related to pattern matching, have been introduced so far. Examples of these structures are *suffix trees* and *directed acyclic word graphs* (*DAWGs*), which can be efficiently constructed in linear time and space. *Compact directed acyclic word graphs* (*CDAWGs*) are an index structure preserving some features of both suffix trees and DAWGs, and require less space than both of them. An algorithm which directly constructs CDAWGs in linear time and space was first introduced by Crochemore and Vérin, based on McCreight's algorithm for constructing suffix trees. In this work, we present a novel *on-line* linear-time algorithm that builds the CDAWG for a single string as well as for a set of strings, inspired by Ukkonen's on-line algorithm for constructing suffix trees.

Key words: pattern matching on strings, compact directed acyclic word graphs, directed acyclic word graphs, suffix trees, on-line and linear-time algorithms

Email addresses: s-ine@i.kyushu-u.ac.jp (Shunsuke Inenaga), hoshino@i.kyushu-u.ac.jp (Hiromasa Hoshino), ayumi@i.kyushu-u.ac.jp (Ayumi Shinohara), takeda@i.kyushu-u.ac.jp (Masayuki Takeda), arikawa@i.kyushu-u.ac.jp (Setsuo Arikawa), mauri@disco.uminib.it (Giancarlo Mauri), pavesi@disco.uminib.it (Giulio Pavesi).

1 Introduction

Several different string problems, like those deriving from the analysis of biological sequences, can be efficiently solved by means of a suitable index structure [11,12,16,10]. The most widely known and studied structure of this kind seems to be *suffix trees* [36,30,6,34,27,33] perhaps because of their myriad applications [1]. For any string w the suffix tree of w requires only $O(n)$ space, and can be built in $O(n)$ time for a fixed alphabet, where n is the length of w. Although its theoretical space complexity is linear, much attention has been devoted to reduction of the practical space requirement of the structure. This has led to the introduction of more space-economical index structures like *suffix arrays* [14,29], *suffix cacti* [25], *compact suffix arrays* [28], *compressed suffix arrays* [15,31], and so on.

Blumer et al. [4] introduced *directed acyclic word graphs* (*DAWG*). In [9] Crochemore pointed out that the DAWG of a string w is the smallest finite state automaton to recognize all suffixes of w. The smallest automaton accepting all factors of a string w, which is a variant of the DAWG for w, was also introduced in [4] and [9]. DAWGs are involved in several combinatorial algorithms on strings [5,17,3,35], since they serve as indexes of the string, as well as other index structures such as suffix tries and suffix trees. Some relationship between suffix trees and DAWGs can be found in [2].

In this work, we focus our attention on *compact directed acyclic word graphs* (*CDAWGs*) first introduced by Blumer et al. [5]. Crochemore and Vérin displayed an overview relationship among suffix tries, suffix trees, DAWGs, and CDAWGs [13]. Suffix trees (resp. DAWGs) are the compacted (resp. minimized) version of suffix tries, as shown in Fig. 1. Similarly, CDAWGs can be obtained by either compacting DAWGs or minimizing suffix trees.

Not only in theory as stated above, but also in practice, do CDAWGs provide significant reductions of the memory space required by suffix trees and DAWGs, as experimental results have shown [5,13]. In Bioinformatics a considerable amount of DNA sequences has to be processed efficiently, both in space and in time. Therefore, from a practical viewpoint, CDAWGs could also play an important role in Bioinformatics.

The first algorithm to construct the CDAWG for a given string w was presented by Blumer et al. [4]. It once builds the DAWG of w, then removes every node of out-degree one and modifies its edges accordingly, so that the resulting structure becomes the CDAWG for w. It runs in liner time, but its main drawback is the construction of the DAWG as an intermediate structure, which takes larger space. A solution to this matter was provided by Crochemore and Vérin [13]: a linear-time algorithm to construct the CDAWG for a string

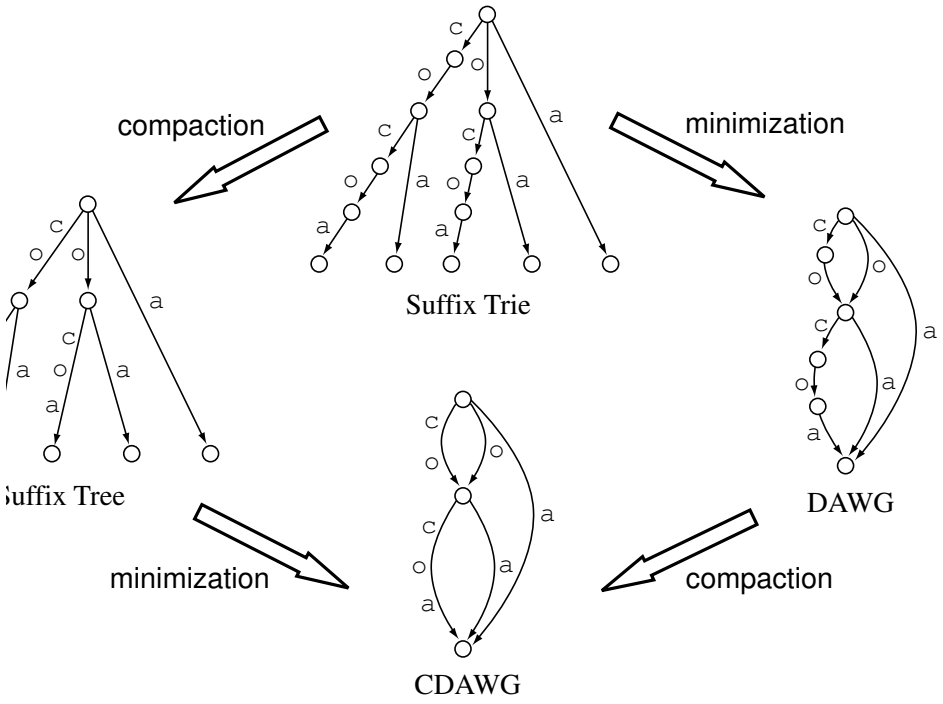

Fig. 1. Relationship among the suffix trie, the suffix tree, the DAWG, and the CDAWG for string `cocoa`.

directly. Their algorithm is based on McCreight's suffix tree construction algorithm [30]. Both algorithms are *off-line*, that is, the whole input string has to be known beforehand. Thus, the structure (suffix tree or CDAWG) has to be rebuilt from scratch, when a new character is added to the input string. Table 1 summarizes some properties of typical algorithms to construct index structures. As seen there, a missing piece, which we have been looking for, is an *on-line algorithm for constructing CDAWGs*.

In this paper, we present a new linear-time algorithm to directly construct the CDAWG for a given string, which is based on Ukkonen's suffix tree construction algorithm [34]. Our algorithm is *on-line*: it processes the characters of the input string from left to right, one by one, with no need to know the whole string beforehand. Our algorithm would be more efficient than the one in [13], in the sense that our algorithm allows us to update the input string. Furthermore, we show that the algorithm can be easily applied to building the CDAWG *for a set of strings*. Actually, the CDAWG for a set of strings can be constructed by the algorithm given in [5] which compacts the DAWG for the set. However, the drawback of this approach is that, when a new string is added to the set, the DAWG has to be built from scratch. Instead, our algorithm permits us the addition of a new string to the set. A previous version of this work appeared in [23].

The rest of the paper is organized as follows. Section 2 is devoted to introduction of basic notation and equivalence relations on strings. In Section 4 we give

Index	*Algorithm*	*linear time*	*on-line*	*multi strings*
suffix tries	Ukkonen [34]		√	√
suffix trees	Weiner [36]	√		
	McCreight [30]	√		
	Ukkonen [34]	√	√	√
DAWGs	Blumer et al. [4]	√	√	
	Crochemore [9]	√	√	
	Blumer et al. [5]	√	√	√
CDAWGs	Blumer et al. [5]	√		√
	Crochemore and Vérin [13]	√		

Table 1
Properties of algorithms to construct index structures.

formal definitions of suffix tries, suffix trees, DAWGs, and CDAWGs, based on the equivalence relations on strings. The definitions give us a good 'unified' view for those index structures, revealing their relationship on the basis of the equivalence relations. We dedicate Section 5 to recalling the on-line algorithm to construct suffix tries given by Ukkonen in [34], which reminds us what an on-line algorithm is like. Ukkonen's on-line suffix tree construction algorithm is also revisited in comparison to the suffix trie algorithm in Section 5. Section 6 introduces our new on-line algorithm that constructs the CDAWG for a given string. It is followed by Section 7 where it is shown that the CDAWG for a set of strings can also be built by the algorithm of Section 6. We finally conclude in Section 8.

2 Preliminaries

2.1 Notation

Let Σ be a finite alphabet. An element of Σ^* is called a *string*. Let x be a string such that $x = a_1 a_2 \cdots a_n$ where $n \geq 1$ and $a_i \in \Sigma$ for $1 \leq i \leq n$. The *length* of x is n and denoted by $|x|$, that is, $|x| = n$. If $n = 0$, x is said to be the *empty string*. It is denoted by ε, that is, $|\varepsilon| = 0$. Let $\Sigma^+ = \Sigma^* - \{\varepsilon\}$.

Strings x, y, and z are said to be a *prefix*, *factor*, and *suffix* of string $w = xyz$, respectively. The sets of the prefixes, factors, and suffixes of a string w are denoted by $\mathit{Prefix}(w)$, $\mathit{Factor}(w)$, and $\mathit{Suffix}(w)$, respectively.

Let w be a string and $|w| = n$. The i-th character of w is denoted by $w[i]$ for $1 \leq i \leq n$, and the factor of w that begins at position i and ends at position j is denoted by $w[i : j]$ for $1 \leq i \leq j \leq n$. For convenience, let $w[i : j] = \varepsilon$ for

$j < i$.

For a set S of strings $w_1, w_2, \ldots, w_\ell$, let $|S|$ denote the cardinality of S, namely, $|S| = \ell$. We denote by $\|S\|$ the total length of strings in S.

The sets of prefixes, factors, and suffixes of the strings in S are denoted by $\mathit{Prefix}(S)$, $\mathit{Factor}(S)$, and $\mathit{Suffix}(S)$, respectively.

Definition 1 *Let $S = \{w_1, \ldots, w_k\}$ where $w_i \in \Sigma^*$ for $1 \leq i \leq k$ and $k \geq 1$. We say that S has the* prefix property *iff $w_i \notin \mathit{Prefix}(w_j)$ for any $1 \leq i \neq j \leq k$.*

3 Equivalence Relations on Strings

Let $S \subseteq \Sigma^*$. For any string $x \in \Sigma^*$, let $Sx^{-1} = \{u \mid ux \in S\}$ and $x^{-1}S = \{u \mid xu \in S\}$.

Definition 2 *Let $w \in \Sigma^*$. The equivalence relations $\equiv_w^L$ and $\equiv_w^R$ on Σ^* are defined by*

$$x \equiv_w^L y \Leftrightarrow \mathit{Prefix}(w)x^{-1} = \mathit{Prefix}(w)y^{-1},$$

$$x \equiv_w^R y \Leftrightarrow x^{-1}\mathit{Suffix}(w) = y^{-1}\mathit{Suffix}(w).$$

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. $[x]_w^R$).

Note that all strings that are not in $\mathit{Factor}(w)$ form one equivalence class under $\equiv_w^L$. This equivalence class is called the *degenerate* class. All other classes are called *non-degenerate*. Similar arguments hold for $\equiv_w^R$.

Proposition 1 (Blumer et al. [5]) *Let $w \in \Sigma^*$ and $x, y \in \mathit{Factor}(w)$. If $x \equiv_w^L y$, then either x is a prefix of y, or vice versa. If $x \equiv_w^R y$, then either x is a suffix of y, or vice versa.*

Definition 3 *For any string $x \in \mathit{Factor}(w)$, $\overrightarrow{x}^w$ (resp. $\overleftarrow{x}^w$) denotes the longest member of $[x]_w^L$ (resp. $[x]_w^R$). We call $\overrightarrow{x}^w$ (resp. $\overleftarrow{x}^w$) the* representative *of $[x]_w^L$ (resp. $[x]_w^R$).*

What $\overrightarrow{x}^w$ (resp. $\overleftarrow{x}^w$) means intuitively is that $\overrightarrow{x}^w$ (resp. $\overleftarrow{x}^w$) is the string obtained by extending x in $[x]_w^L$ (resp. $[x]_w^R$) as long as possible. Each equivalence class in $\equiv_w^L$ ($\equiv_w^R$) other than the degenerate class has a unique longest member. More formally:

Proposition 2 (Inenaga et al. [21], Inenaga [18]) *Let $w \in \Sigma^*$. For any string $x \in \mathit{Factor}(w)$, there uniquely exist strings $\alpha, \beta \in \Sigma^*$ such that $\overrightarrow{x}^w = x\alpha$*

and $\overleftarrow{x}^{w} = \beta x$.

Example 1 *Let* $w = \mathtt{coco}$. *Then* $\overrightarrow{\varepsilon}^{w} = \varepsilon$, $\overrightarrow{\mathtt{c}}^{w} = \overrightarrow{\mathtt{co}}^{w} = \mathtt{co}$, $\overrightarrow{\mathtt{coc}}^{w} = \overrightarrow{\mathtt{coco}}^{w} = \mathtt{coco}$, $\overrightarrow{\mathtt{o}}^{w} = \mathtt{o}$, *and* $\overrightarrow{\mathtt{oc}}^{w} = \overrightarrow{\mathtt{oco}}^{w} = \mathtt{oco}$.

Example 2 *Let* $w = \mathtt{coco}$. *Then* $\overleftarrow{\varepsilon}^{w} = \varepsilon$, $\overleftarrow{\mathtt{c}}^{w} = \mathtt{c}$, $\overleftarrow{\mathtt{o}}^{w} = \overleftarrow{\mathtt{co}}^{w} = \mathtt{co}$, $\overleftarrow{\mathtt{oc}}^{w} = \overleftarrow{\mathtt{coc}}^{w} = \mathtt{coc}$, *and* $\overleftarrow{\mathtt{oco}}^{w} = \overleftarrow{\mathtt{coco}}^{w} = \mathtt{coco}$.

Proposition 3 (Inenaga [18]) *Let* $w \in \Sigma^*$ *and* $x \in Factor(w)$. *Assume* $\overrightarrow{x}^{w} = x$. *Then, for any* $y \in Suffix(x)$, $\overrightarrow{y}^{w} = y$.

We are now introducing a new equivalence class derived from $\overrightarrow{(\cdot)}^{w}$ and $\overleftarrow{(\cdot)}^{w}$.

Definition 4 *For any string* $x \in Factor(w)$, *let* $\overleftrightarrow{x}^{w}$ *be the string* $\beta x \alpha$ *(*$\alpha, \beta \in \Sigma^*$*) such that* $\overrightarrow{x}^{w} = x\alpha$ *and* $\overleftarrow{x}^{w} = \beta x$.

What $\overleftrightarrow{x}^{w} = \beta x \alpha$ implies is that:

(1) every time x occurs in w, it is preceded by β and followed by α.
(2) α and β are the longest strings satisfying (1).

Definition 5 *Let* $x, y \in \Sigma^*$. *We write* $x \equiv_w y$ *if,*

(1) $x, y \in Factor(w)$ *and* $\overleftrightarrow{x}^{w} = \overleftrightarrow{y}^{w}$, *or*
(2) $x \notin Factor(w)$ *and* $y \notin Factor(w)$.

The equivalence class of a string $x \in \Sigma^*$ *with respect to* $\equiv_w$ *is denoted by* $[x]_w$.

For any string $x \in Factor(w)$, $\overleftrightarrow{x}^{w}$ is the unique longest member of $[x]_w$, and is called the *representative* of $[x]_w$.

Example 3 *Let* $w = \mathtt{coco}$. *Then* $\overleftrightarrow{\varepsilon}^{w} = \varepsilon$, $\overleftrightarrow{\mathtt{c}}^{w} = \overleftrightarrow{\mathtt{co}}^{w} = \overleftrightarrow{\mathtt{o}}^{w} = \mathtt{co}$, *and* $\overleftrightarrow{\mathtt{coc}}^{w} = \overleftrightarrow{\mathtt{coco}}^{w} = \overleftrightarrow{\mathtt{oc}}^{w} = \overleftrightarrow{\mathtt{oco}}^{w} = \mathtt{coco}$.

Lemma 1 (Blumer et al. [5]) *The equivalence relation* $\equiv_w$ *is the transitive closure of the relation* $\equiv_w^L \cup \equiv_w^R$.

It follows from the lemma above that:

Corollary 1 *For any string* $x \in Factor(w)$,

$$\overleftrightarrow{x}^{w} = (\overrightarrow{\overleftarrow{x}^{w}}^{w}) = (\overleftarrow{\overrightarrow{x}^{w}}^{w}).$$

Note that, for a string $w \in \Sigma^*$, $|Factor(w)| = O(|w|^2)$. For example, consider string $a^n b^n$. However, considering set $S = \{x \mid x \in Factor(w) \text{ and } x = \overrightarrow{x}^w\}$, we have $|S| = O(|w|)$ for any $w \in \Sigma^*$. Similar arguments hold with $\overleftarrow{x}^w$ and $\overleftrightarrow{x}^w$. The following lemma gives tighter upper-bounds.

Lemma 2 (Blumer et al. [4,5]) *Assume that $|w| > 1$. The number of non-degenerate equivalence classes in $\equiv_w^L$ (or $\equiv_w^R$) is at most $2|w| - 1$. The number of non-degenerate equivalence classes in $\equiv_w$ is at most $|w| + 1$.*

4 Index Structures for Text Strings

In this section, we recall four index structures, the suffix trie, the suffix tree, the directed acyclic word graph (DAWG), and the compact directed acyclic word graph (CDAWG) for a string $w \in \Sigma^*$ denoted by $STrie(w)$, $STree(w)$, $DAWG(w)$, and $CDAWG(w)$, respectively. All these structures represent every string $x \in Factor(w)$. 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. We also define the *suffix links* of each index structure. Suffix links are kinds of failure function often utilized for time-efficient construction of the index structures [36,30,34,4,5,13].

4.1 Suffix Tries

Definition 6 *$STrie(w)$ is the tree (V, E) such that*

$$V = \{x \mid x \in Factor(w)\},$$
$$E = \{(x, a, xa) \mid x, xa \in Factor(w) \text{ and } a \in \Sigma\},$$

and its suffix links are elements of the set

$$F = \{(ax, x) \mid x, ax \in Factor(w) \text{ and } a \in \Sigma\}.$$

Each string $x \in Factor(w)$ has a one-to-one correspondence to a certain node in $STrie(w)$. $STrie(\texttt{coco})$ and $STrie(\texttt{cocoa})$ are displayed in Fig. 2 together with their suffix links.

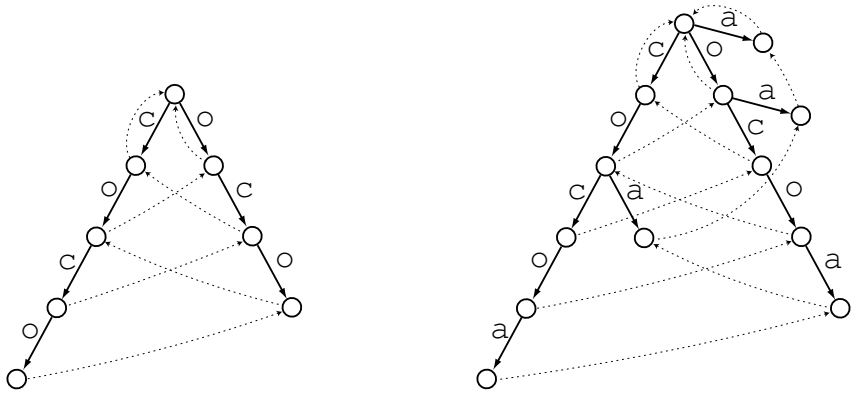

Fig. 2. *STrie*(`coco`) on the left, and *STrie*(`cocoa`) on the right. The solid arrows represent the edges, while the dotted arrows denote the suffix links.

4.2 Suffix Trees

Definition 7 *STree*(w) *is the tree* (V, E) *such that*

$$V = \{\overset{w}{\overrightarrow{x}} \mid x \in \mathit{Factor}(w)\},$$
$$E = \left\{ (\overset{w}{\overrightarrow{x}}, a\beta, \overset{w}{\overrightarrow{xa}}) \;\middle|\; \begin{array}{l} x, xa \in \mathit{Factor}(w),\ a \in \Sigma,\ \beta \in \Sigma^*, \\ \overset{w}{\overrightarrow{xa}} = xa\beta,\ \mathit{and}\ \overset{w}{\overrightarrow{x}} \neq \overset{w}{\overrightarrow{xa}} \end{array} \right\},$$

and its suffix links are elements of the set

$$F = \{(\overset{w}{\overrightarrow{ax}}, \overset{w}{\overrightarrow{x}}) \mid x, ax \in \mathit{Factor}(w),\ a \in \Sigma,\ \mathit{and}\ \overset{w}{\overrightarrow{ax}} = a \cdot \overset{w}{\overrightarrow{x}}\}.$$

The node set of *STree*(w) is a subset of that of *STrie*(w), as seen in the definitions. It means that a string in *Factor*(w) might be represented on an edge in *STree*(w). In this case, we say that the string is represented by an *implicit* node. Conversely, every string in the node set V of *STree*(w) is said to be represented by an *explicit* node. For example, in *STree*(`coco`) of Fig. 3, string `c` is represented by an implicit node, while string `co` is on an explicit node.

STree(w) can be seen as the compacted version of *STrie*(w) with "$\overset{w}{\overrightarrow{(\cdot)}}$ operation". See *STrie*(`cocoa`) in Fig. 2 and *STree*(`cocoa`) in Fig. 3.

Theorem 1 (McCreight [30]) *Let* $w \in \Sigma^*$ *with* $|w| > 1$. *Let* *STree*$(w) = (V, E)$. *Then* $|V| \leq 2|w| - 1$ *and* $|E| \leq 2|w| - 2$.

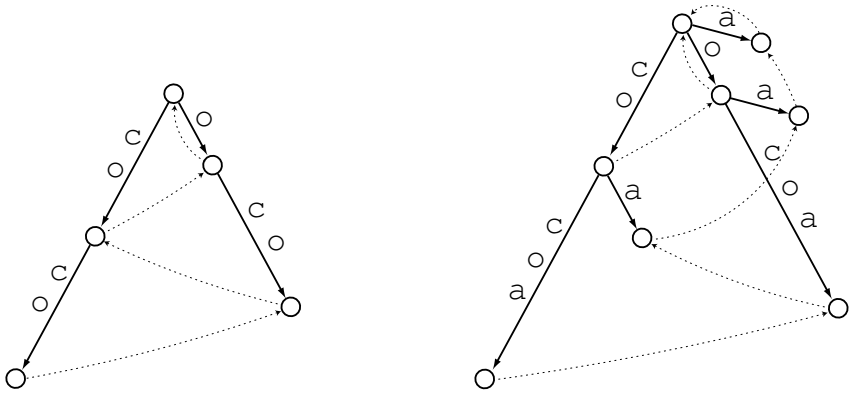

Fig. 3. *STree*(`coco`) on the left, and *STree*(`cocoa`) on the right. The solid arrows represent the edges, while the dotted arrows denote the suffix links.

4.3 DAWGs

Definition 8 *DAWG*(w) *is the directed acyclic graph* (V, E) *such that*

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

and its suffix links are elements of the set

$$F = \{([ax]_w^R, [x]_w^R) \mid x, ax \in \mathit{Factor}(w),\ a \in \Sigma,\ \text{and } [ax]_w^R \neq [x]_w^R\}.$$

The node $[\varepsilon]_w^R$ is called the *source* node of *DAWG*(w). A node of out-degree zero is called a *sink* node of *DAWG*(w). *DAWG*(w) has exactly one sink node for any $w \in \Sigma^*$. We define the *length* of a node $[x]_w^R$ by $|\overleftarrow{x}^w|$.

As seen in the definition, each node of *DAWG*(w) is a non-degenerate equivalence class with respect to $\equiv_w^R$. In Fig. 4, one can see that nodes of *STrie*(`cocoa`) are 'merged' in *DAWG*(`cocoa`) by the equivalence class under $\equiv_w^R$. In this sense, *DAWG*(w) can be seen as the minimized version of *STrie*(w) with "$[(\cdot)]_w^R$ operation".

Theorem 2 (Blumer et al. [4]) *Let* $w \in \Sigma^*$ *with* $|w| > 1$. *Let* *DAWG*$(w) = (V, E)$. *Then* $|V| \leq 2|w| - 1$ *and* $|E| \leq 3|w| - 3$.

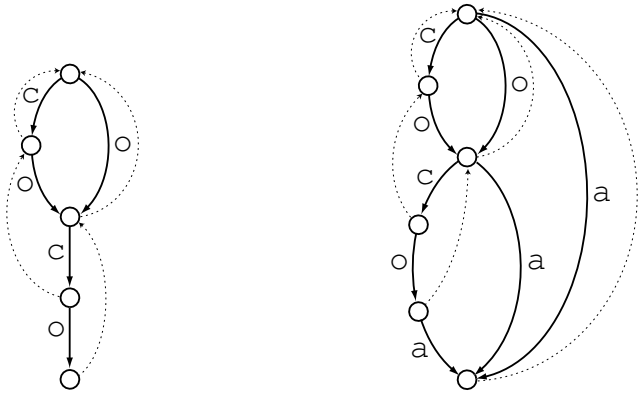

Fig. 4. *DAWG*(coco) on the left, and *DAWG*(cocoa) on the right. The solid arrows represent the edges, while the dotted arrows denote the suffix links.

4.4 CDAWGs

Definition 9 *CDAWG*(w) *is the directed acyclic graph* (V, E) *such that*

$$V = \{[\overrightarrow{x}]^R_w \mid x \in \mathit{Factor}(w)\},$$
$$E = \left\{ ([\overrightarrow{x}]^R_w, a\beta, [\overrightarrow{xa}]^R_w) \;\middle|\; \begin{array}{l} x, xa \in \mathit{Factor}(w),\ a \in \Sigma,\ \beta \in \Sigma^*, \\ \overrightarrow{xa} = xa\beta,\ \mathit{and}\ \overrightarrow{x} \neq \overrightarrow{xa} \end{array} \right\},$$

and its suffix links are elements of the set

$$F = \left\{ ([\overrightarrow{ax}]^R_w, [\overrightarrow{x}]^R_w) \;\middle|\; \begin{array}{l} x, ax \in \mathit{Factor}(w),\ a \in \Sigma, \\ \overrightarrow{ax} = a \cdot \overrightarrow{x},\ \mathit{and}\ [\overrightarrow{x}]^R_w \neq [\overrightarrow{ax}]^R_w \end{array} \right\}.$$

The node $[\overrightarrow{\varepsilon}]^R_w$ is called the *source* node of *CDAWG*(w). A node of out-degree zero is called a *sink* node of *CDAWG*(w). *CDAWG*(w) has exactly one sink node for any $w \in \Sigma^*$. We define the *length* of a node $[\overrightarrow{x}]^R_w$ by $\left|\overleftarrow{(\overrightarrow{x})}\right| = |\overleftrightarrow{x}|$.

It follows from the definition that *CDAWG*(w) is the minimization of *STree*(w) with "$[(\cdot)]^R_w$ operation". In fact, *CDAWG*(cocoa) in Fig. 5 can be obtained by 'merging' the isomorphic subtrees in *STree*(cocoa). *CDAWG*(w) can also be seen as the compaction of *DAWG*(w) with "$\overrightarrow{(\cdot)}$ operation", as seen in *DAWG*(cocoa) in Fig. 4 and *CDAWG*(cocoa).

Theorem 3 (Blumer et al. [5], Crochemore and Vérin [13])
Let $w \in \Sigma^*$ *with* $|w| > 1$. *Let* $\mathit{CDAWG}(w) = (V, E)$. *Then* $|V| \leq |w| + 1$ *and* $|E| \leq 2|w| - 2$.

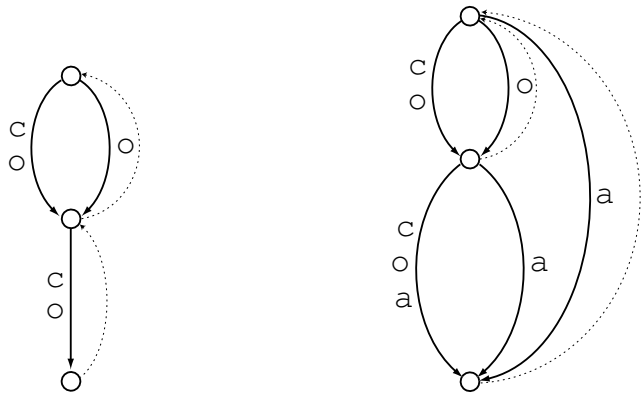

Fig. 5. *CDAWG*(coco) on the left, and *CDAWG*(cocoa) on the right. The solid arrows represent the edges, while the dotted arrows denote the suffix links.

5 General Idea of On-Line Construction Algorithms

5.1 On-Line Construction of Suffix Tries

In this section we recall the on-line algorithm to construct suffix tries given by Ukkonen [34], which is so intuitive that one can understand the basic concept of on-line algorithms.

For a string $x \in \mathit{Factor}(w)$, let $\mathit{suf}(x)$ denote the node reachable via the suffix link of the node x. It derives from Definition 6 that $\mathit{suf}(x) = y$ for some $y \in \mathit{Factor}(w)$ such that $x = ay$ for some character $a \in \Sigma$. In case that $x = \varepsilon$, let $\mathit{suf}(\varepsilon) = \perp$ where $\perp$ is an auxiliary node called the *bottom* node. We suppose that there exists edge $(\perp, a, \varepsilon)$ for any character $a \in \Sigma$. Assuming that the bottom node $\perp$ corresponds to the inverse a^{-1} for any character $a \in \Sigma$, edge $(\perp, a, \varepsilon)$ is consistently defined as well as other edges since $a^{-1} \cdot a = \varepsilon$. The auxiliary node $\perp$ allows us to formalize the algorithm avoiding the distinction between the empty suffix and other non-empty suffixes (in other words, between the root node and other nodes). We leave $\mathit{suf}(\perp)$ undefined.

Suppose we have $\mathit{STrie}(w)$ with some $w \in \Sigma^*$. We now consider updating $\mathit{STrie}(w)$ to $\mathit{STrie}(wa)$ with an arbitrary character $a \in \Sigma$. What is necessary here is to insert suffixes of wa into $\mathit{STrie}(w)$.

Definition 10 *For an arbitrary string $w \in \Sigma^*$ and character $a \in \Sigma$, the* longest repeated suffix *(*LRS*) of wa is the longest element of the set $\mathit{Factor}(w) \cap \mathit{Suffix}(wa)$.*

The LRS of string w is denoted by $\mathit{LRS}(w)$.

Example 4 *The LRS of string* coco *is* co*. The LRS of string* cocoa *is ε.*

It is guaranteed that the LRS always exists for any string $w \in \Sigma^*$ since the empty string ε belongs to the set $\mathit{Factor}(w) \cap \mathit{Suffix}(wa)$ for any character $a \in \Sigma$.

Let $w \in \Sigma^*$ with $|w| = n$, and $a \in \Sigma$. Let $u_1, u_2, \ldots, u_n, u_{n+1}, u_{n+2}$ be the suffixes of string wa sorted in decreasing order of their lengths, that is, $u_1 = wa$ and $u_{n+2} = \varepsilon$. These suffixes are divided into the following two groups as follows, by $\mathit{LRS}(wa)$.

(1) $u_1, \ldots, u_\ell$ where $u_{\ell+1} = \mathit{LRS}(wa)$.
(2) $u_{\ell+1}, \ldots, u_{n+2}$.

It follows from the definition of $\mathit{LRS}(wa)$ that the suffixes in the group (2) are already represented in $\mathit{STrie}(wa)$. Therefore, there is no need to newly insert any suffixes of the group (2) into $\mathit{STrie}(w)$.

Let $v_1, \ldots, v_{n+1}$ be the suffixes of w sorted in decreasing order of their lengths, that is, $v_1 = w$ and $v_{n+1} = \varepsilon$. Notice that $v_k a = u_k$ for any k with $1 \leq k \leq n+1$. The on-line algorithm starts with inserting into $\mathit{STrie}(w)$ the longest suffix u_1 of wa, which is wa itself. To do so, the algorithm creates a new edge labeled with a from the node $v_1 = w$ to a new node, which turns out to represent $wa = u_1$. It then focuses on the node that represents v_2. Note that it is feasible in constant time to move from node v_1 to node v_2 by using the suffix link $\mathit{suf}(v_1)$. We then create a new edge (v_2, a, u_2), and the suffix link of node u_1 is connected to u_2, namely, $\mathit{suf}(u_1) = u_2$. We continue this operation until encountering $\mathit{LRS}(wa) = u_{\ell+1}$, and the resulting structure is $\mathit{STrie}(wa)$. The node with respect to $\mathit{LRS}(wa) = u_{\ell+1}$ is called the *end point* of $\mathit{STrie}(wa)$. This way suffix tries can be constructed in on-line fashion. However, since the space requirement of $\mathit{STrie}(w)$ is quadratic in $|w|$, this on-line algorithm does not run in linear time.

Theorem 4 (Ukkonen [34]) *Assume Σ is a fixed alphabet. For any string $w \in \Sigma^*$, $\mathit{STrie}(w)$ can be constructed on-line and in $O(|w|^2)$ time, using $O(|w|^2)$ space.*

On-line construction of $\mathit{STrie}(\texttt{cocoa})$ is illustrated in Fig. 6.

5.2 On-Line Construction of Suffix Trees

This section is devoted to recalling Ukkonen's on-line algorithm to construct suffix trees [34].

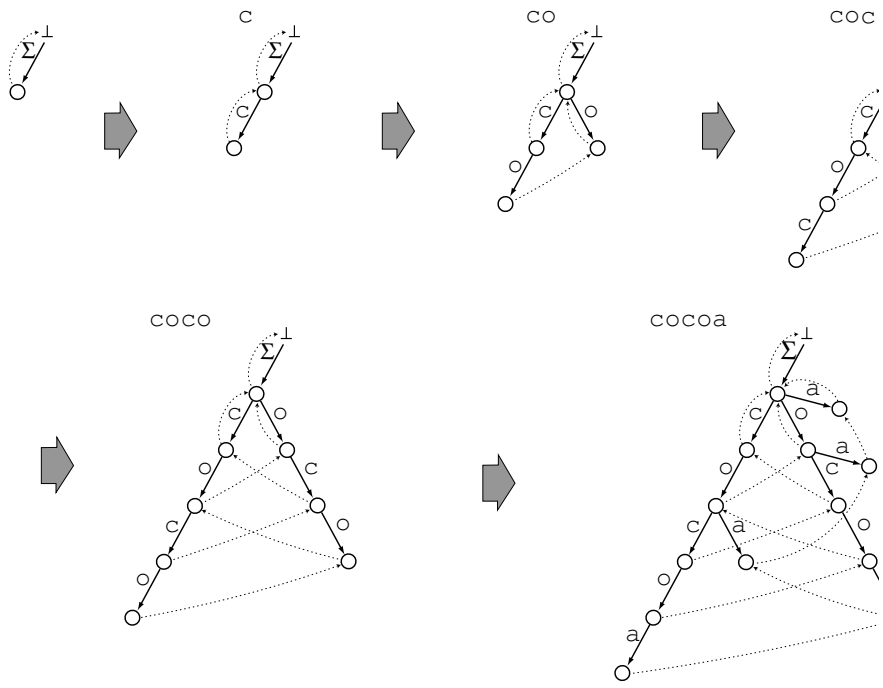

Fig. 6. On-line construction of $STrie(w)$ with $w = \texttt{cocoa}$.

5.2.1 Suffix Trees Redefined

For any string w, let $STree'(w)$ denote the tree obtained by removing all internal nodes of out-degree one from $STree(w)$. $STree'(\texttt{coco})$ and $STree'(\texttt{cocoa})$ are shown in Fig. 7. Actually, Ukkonen's suffix tree construction algorithm [34] builds $STree'(w)$, not $STree(w)$, since $STree'(w)$ is suitable in on-line string-processing scheme, as to be shown in the sequel.

To give a formal definition of $STree'(w)$, we introduce a relation X_w over Σ^* such that

$$X_w = \{(x, xa) \mid x \in Factor(w) \text{ and } a \in \Sigma \text{ is unique such that } xa \in Factor(w)\},$$

and let $\equiv'^L_w$ be the equivalence closure of X_w, i.e., the smallest superset of X_w that is symmetric, reflexive, and transitive.

Proposition 4 (Inenaga [18]) *For any string $w \in \Sigma^*$, $\equiv^L_w$ is a refinement of $\equiv'^L_w$, namely, every equivalence class under $\equiv'^L_w$ is a union of one or more equivalence classes in $\equiv^L_w$.*

For a string $x \in Factor(w)$, let $\overset{w}{\Longrightarrow}{x}$ denote the longest string in the equivalence class to which x belongs under the equivalence relation $\equiv'^L_w$.

Example 5 *Let $w = \texttt{coco}$. Then $\overset{w}{\Longrightarrow}{\varepsilon} = \varepsilon$, $\overset{w}{\Longrightarrow}{\texttt{c}} = \overset{w}{\Longrightarrow}{\texttt{co}} = \overset{w}{\Longrightarrow}{\texttt{coc}} = \overset{w}{\Longrightarrow}{\texttt{coco}} = \texttt{coco}$, and $\overset{w}{\Longrightarrow}{\texttt{o}} = \overset{w}{\Longrightarrow}{\texttt{oc}} = \overset{w}{\Longrightarrow}{\texttt{oco}} = \texttt{oco}$.*

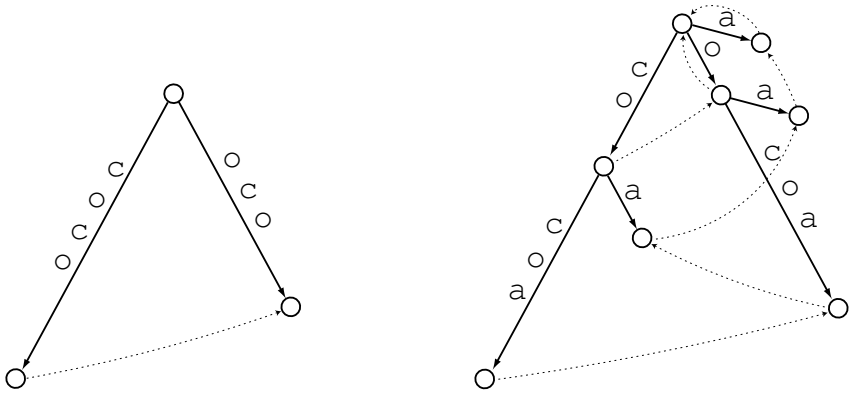

Fig. 7. *STree′*(coco) on the left, and *STree′*(cocoa) on the right. The solid arrows represent the edges, while the dotted arrows denote the suffix links.

Compare the above example with Example 1 for $\overrightarrow{(\cdot)}^{w}$.

Proposition 5 (Inenaga [18]) *For any string* $x \in \Sigma^*$, $\overrightarrow{x}^{w}$ *is a prefix of* $\overrightarrow{\overrightarrow{x}}^{w}$. *If* $\overrightarrow{x}^{w} \neq \overrightarrow{\overrightarrow{x}}^{w}$, *then* $\overrightarrow{x}^{w} \in \mathit{Suffix}(w)$.

Proposition 6 (Inenaga [18]) *If set* $\mathit{Suffix}(w) - \{\varepsilon\}$ *satisfies the prefix property,* $\overrightarrow{\overrightarrow{x}}^{w} = \overrightarrow{x}^{w}$ *for any string* $x \in \mathit{Factor}(w)$.

We are now ready to define *STree′*(w).

Definition 11 *STree′*(w) *is the tree* (V, E) *such that*

$$V = \{\overrightarrow{\overrightarrow{x}}^{w} \mid x \in \mathit{Factor}(w)\},$$

$$E = \left\{ (\overrightarrow{\overrightarrow{x}}^{w}, a\beta, \overrightarrow{\overrightarrow{xa}}^{w}) \;\middle|\; \begin{array}{l} x, xa \in \mathit{Factor}(w),\ a \in \Sigma,\ \beta \in \Sigma^*, \\ \overrightarrow{\overrightarrow{xa}}^{w} = xa\beta,\ \mathit{and}\ \overrightarrow{\overrightarrow{x}}^{w} \neq \overrightarrow{\overrightarrow{xa}}^{w} \end{array} \right\},$$

and its suffix links are elements of the set

$$F = \{(\overrightarrow{\overrightarrow{ax}}^{w}, \overrightarrow{\overrightarrow{x}}^{w}) \mid x, ax \in \mathit{Factor}(w),\ a \in \Sigma,\ \mathit{and}\ \overrightarrow{\overrightarrow{ax}}^{w} = a \cdot \overrightarrow{\overrightarrow{x}}^{w}\}.$$

This definition is the same as the one obtained by replacing "$\overrightarrow{(\cdot)}^{w}$ operation" with "$\overrightarrow{\overrightarrow{(\cdot)}}^{w}$ operation" in Definition 7. Therefore, it follows from Proposition 6 that:

Corollary 2 *If* $\mathit{Suffix}(w) - \{\varepsilon\}$ *has the prefix property,* $\mathit{STree}'(w) = \mathit{STree}(w)$.

Even in case that the set $\mathit{Suffix}(w) - \{\varepsilon\}$ does not have the prefix property,

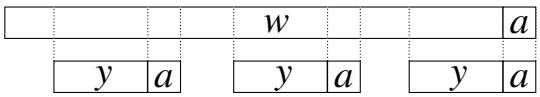

Fig. 8. Illustration for the proof of Lemma 7.

$STree'(w\$)$ is identical to $STree(w\$)$ where \$ is a special character that occurs nowhere in w.

As previously stated, Ukkonen's algorithm constructs $STree'(w)$ since $STree'(w)$ is suitable in on-line manner. The reasons are as follows.

Proposition 7 *Let $w \in \Sigma^*$ and $a \in \Sigma$. Let $z = LRS(w)$. Let $y \in Suffix(z)$. Assume $\overrightarrow{y}^{w} = y$, and there uniquely exists a character $a \in \Sigma$ such that $ya \in Factor(w)$. Then, $\overrightarrow{y}^{wa} = ya$.*

Proof. Since $y \in Suffix(z)$, $ya \in Suffix(za)$. Because a is the only character such that $ya \in Factor(w)$, it is also the only character such that $ya \in Factor(wa)$. Consequently we have $Prefix(wa)(y)^{-1} = Prefix(wa)(ya)^{-1}$, which implies that $y \equiv^{L}_{wa} ya$. It is clear that ya is the longest element of $[ya]^{L}_{wa} = [y]^{L}_{wa}$, meaning that $\overrightarrow{ya}^{wa} = \overrightarrow{y}^{wa} = ya$. (See Fig. 8). □

This proposition implies that an explicit node of $STree(w)$ may become implicit in $STree(wa)$. The maintenance of converting all of such explicit nodes into implicit nodes would take quadratic time in aggregate. This causes difficulty in updating $STree(w)$ into $STree(wa)$ in amortized constant time. However, we have the following proposition for $STree'(w)$.

Proposition 8 *Let $w \in \Sigma^*$ and $a \in \Sigma$. Let $z = LRS(w)$. Let $y \in Suffix(z)$. Assume $\overrightarrow{\overrightarrow{y}}^{w} = y$. Then $\overrightarrow{\overrightarrow{y}}^{wa} = y$ for any $a \in \Sigma$.*

Proof. From the assumption that $\overrightarrow{\overrightarrow{y}}^{w} = y$, there exist at least two distinct characters b, c such that $yb, yc \in Factor(w)$. It is obvious that $yb, yc \in Factor(wa)$ for any $a \in \Sigma$, and thus we have $\overrightarrow{\overrightarrow{y}}^{wa} = y$. □

According to this proposition, in converting $STree'(w)$ into $STree'(wa)$ we do not need the maintenance that is required in converting $STree(w)$ into $STree(wa)$ due to Proposition 7. This is why $STree'(w)$ is suitable for on-line algorithm if we want it to run in linear time.

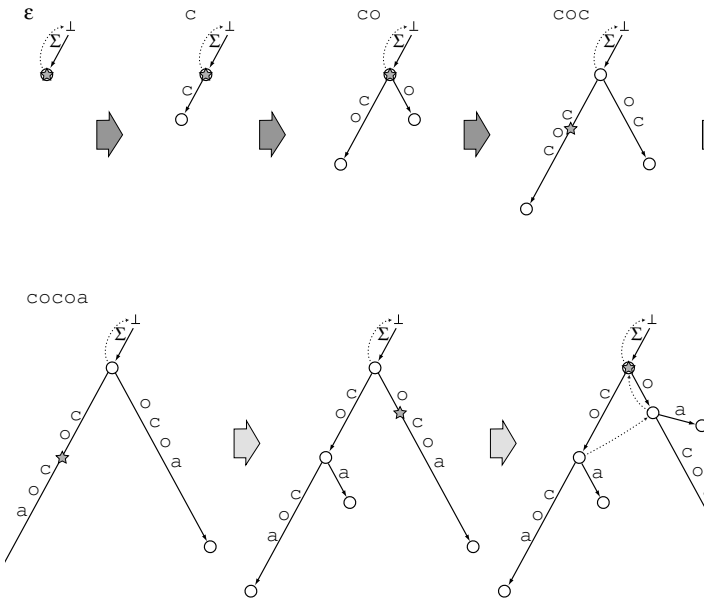

Fig. 9. On-line construction of $STree'(w)$ with $w = \texttt{cocoa}$. The star represents the active point for each step.

5.2.2 Ukkonen's Algorithm

Informal Description of the Algorithm.

Unlike the case of suffix tries mentioned in Section 5.1, Ukkonen's on-line suffix tree construction algorithm runs in linear time.

Theorem 5 (Ukkonen [34]) *Assume Σ is a fixed alphabet. For any string $w \in \Sigma^*$, $STree'(w)$ can be constructed on-line and in $O(|w|)$ time, using $O(|w|)$ space.*

We here summarize Ukkonen's suffix tree construction algorithm in the comparison with the previous suffix trie algorithm. Fig. 9 illustrates on-line construction of $STree'(\texttt{cocoa})$.

Focus on the update of $STree'(\texttt{co})$ to $STree'(\texttt{coc})$. Differently from that of $STrie(\texttt{co})$ to $STrie(\texttt{coc})$, the edges leading to the leaf nodes are *automatically* extended with the new character `c` in $STree'(\texttt{coc})$. This is feasible by the idea of so-called *open edges*.

See the first and second steps of the update of $STree'(\texttt{coco})$ to $STree'(\texttt{cocoa})$. The gray star mark indicates the *active point* from which a new edge is created in each step. After the new edge $(\texttt{co}, \texttt{a}, \texttt{coa})$ is inserted, the active point moves to the implicit node for string `o`. In case of the suffix trie, it is possible to move there by traversing the suffix link of node `co`. However, there is yet to be the suffix link of node `co` in the suffix tree. Thereof, Ukkonen's algorithm *simulates* the traversal of the suffix link as follows: First, it goes up to the explicit parent

node ε of node co which already has its suffix link. After that, it moves to the bottom node $\perp$ via the suffix link of the root node, and then advances along the path spelling out co. (Note that the string co corresponds to the label of the edge the active point went up backward.) This way, in Ukkonen's algorithm the active point moves via 'implicit' suffix links. Since the suffix links of leaf nodes are never utilized in Ukkonen's algorithm, none of them are created.

Formal Description of the Algorithm.

Ukkonen's on-line suffix tree construction algorithm is based on the on-line algorithm to build suffix tries recalled in Section 5.1. As stated in Definition 11, an edge of $\mathit{STree}'(w)$ is labeled by a string $\alpha \in \mathit{Factor}(w)$. The key idea to achieve a linear-space implementation of the suffix tree is to label each edge $(\overset{w}{\Longrightarrow}x, \alpha, \overset{w}{\Longrightarrow}x\alpha)$ in $\mathit{STree}'(w)$ by (k, p), such that $w[k:p] = \alpha$.

An implicit node $y \in \mathit{Factor}(w)$ can be represented by an ordered pair $(\overset{w}{\Longrightarrow}x, \alpha)$ of an explicit node $\overset{w}{\Longrightarrow}x$ and a string $\alpha \in \mathit{Factor}(w)$ such that $y = \overset{w}{\Longrightarrow}x \cdot \alpha$. The ordered pair $(\overset{w}{\Longrightarrow}x, \alpha)$ is called a *reference pair* for the implicit node y. Note that explicit nodes can also be represented by reference pairs. There can be one or more reference pairs for a node y. The reference pair $(\overset{w}{\Longrightarrow}x, \alpha)$ for y in which $|\alpha|$ is minimized is called the *canonical* reference pair for y. The reference pair can also be represented using the integers h, p, as $(\overset{w}{\Longrightarrow}x, (h, p))$ such that $w[h:p] = \alpha$.

Suppose we have $\mathit{STree}'(w)$ with some $w \in \Sigma^*$. We now consider updating $\mathit{STree}'(w)$ to $\mathit{STree}'(wa)$ with an arbitrary character $a \in \Sigma$. What is necessary here is to insert suffixes of wa into $\mathit{STree}'(w)$. The group (1) of the suffixes of wa, mentioned in the previous section, can moreover be divided into two sub-groups as follows, by $\mathit{LRS}(w)$.

(1-a) $u_1, \ldots, u_k$ where $u_{k+1} = \mathit{LRS}(w) \cdot a$.
(1-b) $u_{k+1}, \ldots, u_\ell$.

We remark that all the suffixes of the group (1-a) are those represented by the leaf nodes in $\mathit{STree}'(w)$. Note that, for any i with $1 \leq i \leq k$, we have $\overset{w}{\Longrightarrow}v_i = v_i$ and $\overset{wa}{\Longrightarrow}u_i = u_i$. Moreover, $v_i a = u_i$ (see Section 5.1). This means that, intuitively, every leaf node of $\mathit{STree}'(w)$ is also a leaf node in $\mathit{STree}'(wa)$. This fact is crucial to Ukkonen's algorithm in order that it is able to *automatically* insert the suffixes in the group (1-a) into $\mathit{STree}'(wa)$, by means of *open edges* which will be recalled below. Suppose that $(\overset{w}{\Longrightarrow}x, \alpha, \overset{w}{\Longrightarrow}x\alpha)$ is an edge of $\mathit{STree}'(w)$

where $\overset{w}{\Longrightarrow}\!\!x\alpha$ is a leaf node. Let p be the integer such that $w[p:|w|] = \alpha$. Then we actually implement the edge label by (p, ∞), where its implication is "a leaf node is always a leaf node.". This way we need no explicit insertion of the suffixes of w in the group (1-a).

For a suffix u_j of group (1-b) where $k+1 \leq j \leq \ell$, the location in $STree'(w)$ from which u_j is inserted is called the *active point* of u_j in $STree'(w)$. Note that the active point for u_{k+1} is the end point of $STree'(w)$. As well as the case of on-line construction of suffix tries, we insert the suffixes $u_{k+1}, \ldots, u_\ell$ of wa into $STree'(w)$ in decreasing order of their lengths. To do so, we focus on the locations of the suffixes $v_{k+1}, \ldots, v_\ell$ of w from which a new edge labeled by the new character a is created. Assume we are now inserting suffix u_j into $STree'(w)$ for some $k+1 \leq j \leq \ell$. There are the two following cases regarding the active point.

(Case 1) The active point is on an explicit node v_j. In this case,

$$\overset{w}{\Longrightarrow}\!\!v_j = \overset{wa}{\Longrightarrow}\!\!v_j = v_j.$$

Then a new edge $(\overset{wa}{\Longrightarrow}\!\!v_j, \alpha, \overset{wa}{\Longrightarrow}\!\!v_j\alpha)$ is created, where α is a at the moment. Note $\overset{wa}{\Longrightarrow}\!\!v_j\alpha = \overset{wa}{\Longrightarrow}\!\!v_ja = v_ja = u_j$. In order for the implementation of an open edge, the edge is actually labeled by $(n+1, \infty)$, where $|wa| = n+1$ and $wa[n+1] = a$. After that, the active point moves to the explicit node $suf(\overset{w}{\Longrightarrow}\!\!v_j) = \overset{w}{\Longrightarrow}\!\!v_{j+1} = v_{j+1}$, in order to insert the next suffix u_{j+1} of wa.

(Case 2) The active point is on an implicit node v_j. In this case,

$$\overset{w}{\Longrightarrow}\!\!v_j \neq v_j \text{ but } \overset{wa}{\Longrightarrow}\!\!v_j = v_j.$$

Let $(\overset{w}{\Longrightarrow}\!\!x, \alpha)$ be the canonical reference pair for the active point, namely, $\overset{w}{\Longrightarrow}\!\!x \cdot \alpha = v_j$. Focus on the edge $(\overset{w}{\Longrightarrow}\!\!x, \alpha\beta, \overset{w}{\Longrightarrow}\!\!x\alpha\beta)$ with some non-empty string β. The implicit node representing v_j is located on this edge. The edge is then replaced by (separated into) the edges $(\overset{wa}{\Longrightarrow}\!\!x, \alpha, \overset{wa}{\Longrightarrow}\!\!x\alpha)$ and $(\overset{wa}{\Longrightarrow}\!\!x\alpha, \beta, \overset{wa}{\Longrightarrow}\!\!x\alpha\beta)$ where $\overset{wa}{\Longrightarrow}\!\!x\alpha = v_j$ is a new explicit node. Then a new edge $(\overset{wa}{\Longrightarrow}\!\!x\alpha, \gamma, \overset{wa}{\Longrightarrow}\!\!x\alpha\gamma)$ is created, where $\gamma = a$ at the moment. Note $\overset{wa}{\Longrightarrow}\!\!x\alpha\gamma = \overset{wa}{\Longrightarrow}\!\!v_ja = v_ja = u_j$. In order for the implementation of an open edge, the edge is actually labeled by $(n+1, \infty)$, where $|wa| = n+1$ and $wa[n+1] = a$.

After that, we need to move to the (implicit or explicit) node corresponding to v_{j+1}, the next active point, but the table *suf* is yet to be computed for the node $\overset{wa}{\Longrightarrow}\!\!x\alpha$ that has been created just now. Thus we once move to its parent node $\overset{wa}{\Longrightarrow}\!\!x$ for which $suf(\overset{wa}{\Longrightarrow}\!\!x)$ must have already been computed. Let

$suf(\overset{wa}{\Longrightarrow}{x}) = \overset{wa}{\Longrightarrow}{y}$. Then $\overset{wa}{\Longrightarrow}{y}$ is explicit. Note that $\overset{wa}{\Longrightarrow}{y} \cdot \alpha = v_{j+1}$. We go down from the node $\overset{wa}{\Longrightarrow}{y}$ while spelling out α, to obtain the canonical reference pair for the active point v_{j+1}. The node v_{j+1} either is already, or will in this step become, explicit. $suf(\overset{wa}{\Longrightarrow}{x\alpha}) = suf(\overset{wa}{\Longrightarrow}{v_j})$ is connected to v_{j+1}. This way the algorithm 'simulates' the suffix-link-traversal of suffix tries.

A pseudo-code for Ukkonen's algorithm is shown in Fig. 10. Function *canonize* is a routine to canonize a given reference pair. Function *check_end_point* is one that returns true if a given reference pair is the end point, and false otherwise. Function *split_edge* splits an edge into two, by creating a new explicit node at the position to which the given reference pair corresponds.

6 On-Line Construction of CDAWGs

In this section we give an on-line algorithm to construct CDAWGs. In order to achieve a linear-time algorithm, we shall need CDAWGs that are defined on the basis of $\overset{w}{\Longrightarrow}{(\cdot)}$ rather than $\overset{w}{\longrightarrow}{(\cdot)}$, like in case of suffix trees recalled in Section 5.2.

6.1 CDAWGs Redefined

Definition 12 *CDAWG′(w) is the directed acyclic graph (V, E) such that*

$$V = \{[\overset{w}{\Longrightarrow}{x}]^R_w \mid x \in \mathit{Factor}(w)\},$$
$$E = \left\{ ([\overset{w}{\Longrightarrow}{x}]^R_w, a\beta, [\overset{w}{\Longrightarrow}{xa}]^R_w) \;\middle|\; \begin{array}{l} x, xa \in \mathit{Factor}(w),\ a \in \Sigma,\ \beta \in \Sigma^*, \\ \overset{w}{\Longrightarrow}{xa} = xa\beta,\ \mathit{and}\ \overset{w}{\Longrightarrow}{x} \neq \overset{w}{\Longrightarrow}{xa} \end{array} \right\},$$

and its suffix links are elements of the set

$$F = \left\{ ([\overset{w}{\Longrightarrow}{ax}]^R_w, [\overset{w}{\Longrightarrow}{x}]^R_w) \;\middle|\; \begin{array}{l} x, ax \in \mathit{Factor}(w),\ a \in \Sigma, \\ \overset{w}{\Longrightarrow}{ax} = a \cdot \overset{w}{\Longrightarrow}{x},\ \mathit{and}\ [\overset{w}{\Longrightarrow}{x}]^R_w \neq [\overset{w}{\Longrightarrow}{ax}]^R_w \end{array} \right\}.$$

In Fig. 7 and 11, one can see that nodes of *STree′*(`cocoa`) are 'merged' in *CDAWG′*(`cocoa`) by the equivalence class under $\equiv^R_w$. In this sense *CDAWG′(w)* can be seen as the minimized version of *STree′(w)* with "$[(\cdot)]^R_w$ operation".

```
Algorithm for on-line construction of STree'(w$)
in alphabet Σ = {w[−1], w[−2], ... w[−m]}.
/* $ is the end-marker appearing nowhere in w. */
 1 create nodes root and ⊥;
 2 for j := 1 to m do create edge (⊥, (−j, −j), root);
 3 suf(root) := ⊥;
 4 (s, k) := (root, 1);    i := 0;
 5 repeat
 6     i := i + 1;
 7     (s, k) := update(s, (k, i));
 8 until w[i] = $;

function update(s, (k, p)): pair of integers;
/* (s, (k, p − 1)) is the canonical reference pair for the active point. */
 1 c := w[p];    oldr := nil;
 2 while not check_end_point(s, (k, p − 1), c) do
 3     if k ≤ p − 1 then r := split_edge(s, (k, p − 1));    /* implicit case. */
 4     else r := s;    /* explicit case. */
 5     create node r';    create edge (r, (p, ∞), r');
 6     if oldr ≠ nil then suf(oldr) := r;
 7     oldr := r;
 8     (s, k) := canonize(suf(s), (k, p − 1));
 9 if oldr ≠ nil then suf(oldr) := s;
10 return canonize(s, (k, p));

function check_end_point(s, (k, p), c): boolean;
 1 if k ≤ p then    /* implicit case. */
 2     let (s, (k', p'), s') be the w[k]-edge from s;
 3     return (c = w[k' + p − k + 1]);
 4 else return (there is a c-edge from s);

function canonize(s, (k, p)): pair of node and integers;
 1 if k > p then return (s, k);    /* explicit case. */
 2 find the w[k]-edge (s, (k', p'), s') from s;
 3 while p' − k' ≤ p − k do
 4     k := k + p' − k' + 1;    s := s';
 5     if k ≤ p then find the w[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 w[k]-edge from s;
 2 create node r;
 3 replace the edge by (s, (k', k' + p − k), r) and (r, (k' + p − k + 1, p'), s');
 4 return r;
```

Fig. 10. Ukkonen's on-line algorithm for constructing suffix trees.

The node $[\overrightarrow{\overrightarrow{\varepsilon}}^{w}]^{R}_{w}$ is called the *source* node of $CDAWG'(w)$. A node of out-degree zero is called a *sink* node of $CDAWG'(w)$. $CDAWG'(w)$ has exactly one sink node for any $w \in \Sigma^*$, which is $[\overrightarrow{\overrightarrow{w}}^{w}]^{R}_{w}$. We define the *length* of a node $[\overrightarrow{\overrightarrow{x}}^{w}]^{R}_{w}$ by $\left|(\overleftarrow{\overrightarrow{x}}^{w})\right|$, namely, by the length of the representative of $[\overrightarrow{\overrightarrow{x}}^{w}]^{R}_{w}$. For examples, the

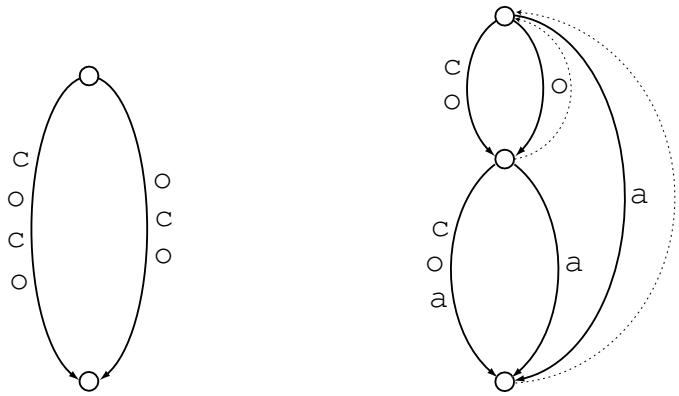

Fig. 11. *CDAWG′*(coco) on the left, and *CDAWG′*(cocoa) on the right. The solid arrows represent the edges, while the dotted arrows denote the suffix links.

length of the source node is 0 since $\left|\overleftarrow{(\overrightarrow{\varepsilon})}^{w}\right| = |\overleftarrow{\varepsilon}^{w}| = |\varepsilon| = 0$, and the length of the sink node is $|w|$ since $\left|\overleftarrow{(\overrightarrow{w})}^{w}\right| = |\overleftarrow{w}^{w}| = |w|$.

Definition 12 equals the one obtained by replacing "$\overrightarrow{(\cdot)}^{w}$ operation" with "$\overrightarrow{(\cdot)}^{w}$ operation" in Definition 9. Therefore, it follows from Proposition 6 that:

Corollary 3 *If set Suffix*$(w) - \{\varepsilon\}$ *has the prefix property, CDAWG′*$(w) =$ *CDAWG*(w).

Even if *Suffix*$(w) - \{\varepsilon\}$ does not have the prefix property, *CDAWG′*$(w\$)$ is identical to *CDAWG*$(w\$)$ where \$ is a special character that occurs nowhere in w.

As well as the case with suffix trees, according to Propositions 8 and 7, *CDAWG′*(w) is more suitable to treat in on-line manner. This is why we gave Definition 12 above, which our on-line algorithm actually constructs.

6.2 Our Algorithm

Informal Description of the Algorithm.

Before delving into the technical detail of the algorithm for on-line construction of CDAWGs, we informally describe how a CDAWG is built on-line. See Fig. 12 that shows on-line construction of *CDAWG′*(cocoa), in comparison with Fig. 9 displaying on-line construction of *STree′*(cocoa). Compare *CDAWG′*(co) and *STree′*(co). While strings co and o are separately represented in *STree′*(co), they are in the same node in *CDAWG′*(co). The desti-

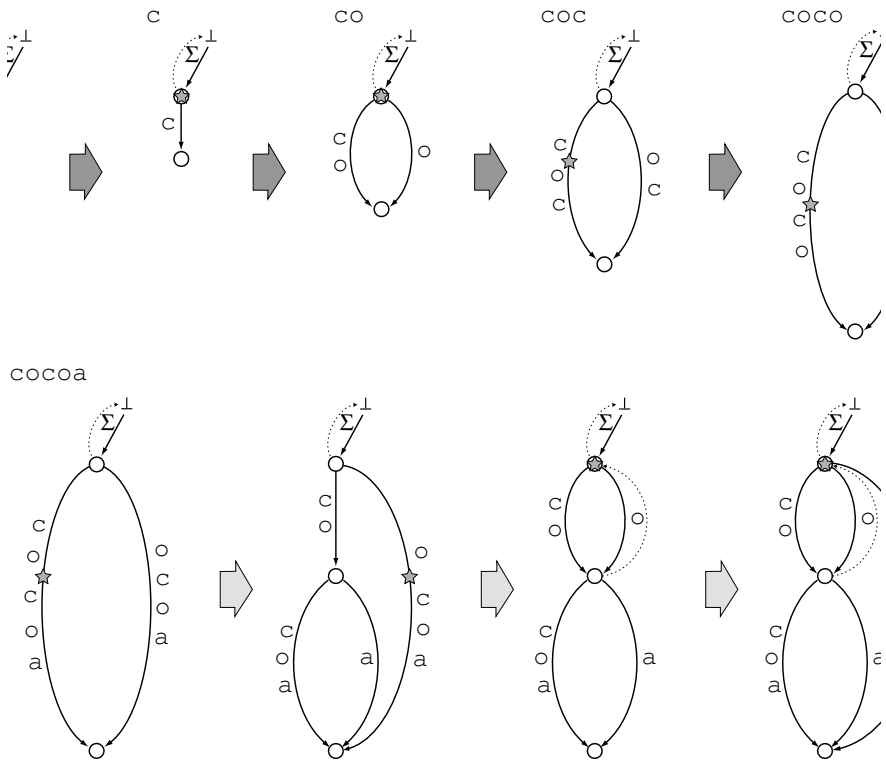

Fig. 12. On-line construction of $CDAWG'(w)$ with $w = \texttt{cocoa}$. The star mark represents the active point for each step.

nation of any open edge of a CDAWG is all the same, the sink node. Open edges of a CDAWG are also *automatically* extended, as well as those of a suffix tree (see $CDAWG'(\texttt{coc})$ and $CDAWG'(\texttt{coco})$).

Focus on the first step of the update of $CDAWG'(\texttt{coco})$ to $CDAWG'(\texttt{cocoa})$. String co there gets to be explicitely represented, and the active point is on implicit node o. In case of the construction of $STree'(\texttt{cocoa})$, edge $(\varepsilon, \texttt{ocoa}, \texttt{ocoa})$ is split into two edges $(\varepsilon, \texttt{o}, \texttt{o})$ and $(\texttt{o}, \texttt{coa}, \texttt{ocoa})$, and then an open edge $(\texttt{o}, \texttt{a}, \texttt{oa})$ is newly created (see Fig. 9). However, in case of the CDAWG, edge $(\varepsilon, \texttt{ocoa}, \texttt{ocoa})$ is *redirected* to node co, and the label is simultaneously modified. Since strings co and o are equivalent under the equivalence relation $\equiv^R_{\texttt{cocoa}}$, they are *merged* into a single node in $CDAWG'(\texttt{cocoa})$.

Formal Description of the Algorithm.

The algorithm presented in this section for on-line construction of CDAWGs behaves similarly to Ukkonen's on-line suffix tree construction algorithm. Let $w \in \Sigma^*$ with $|w| = n$, and $a \in \Sigma$. Remember the set of the suffixes of w are divided into three sub-groups as mentioned in Sections 5.1 and 5.2. Let $u_1, u_2, \ldots, u_n, u_{n+1}, u_{n+2}$ be the suffixes of string wa sorted in decreasing order of their lengths, and let $v_1, v_2, \ldots, v_n, v_{n+1}$ be the suffixes of string w sorted in

decreasing order of their lengths. Namely, $v_i a = u_i$ for any i with $1 \leq i \leq n+1$. Assume $u_{\ell+1} = LRS(wa)$ and $v_{k+1} = LRS(w)$. The difference between on-line suffix tree construction and on-line CDAWG construction is summarized as follows:

- All the suffixes in the group (1) are equivalent under $\equiv^R_{wa}$. Thus all of them are represented by the same node $[\overset{wa}{\Longrightarrow}\overline{u_1}]^R_{wa} = [\overset{wa}{\Longrightarrow}\overline{u_\ell}]^R_{wa}$, which is in fact the sink node of $CDAWG'(wa)$. Namely, the destinations of the open edges are all the same. Moreover, $\overset{w}{\Longrightarrow}\overline{v_i} = v_i$ and $\overset{wa}{\Longrightarrow}\overline{u_i} = u_i$ for any i with $1 \leq i \leq k$. According to these properties, we can generalize the idea of open edges as follows. For any open edge $(s, (p, \infty), t)$ of $CDAWG'(w)$ where t denotes the sink node, we actually implement it as $(s, (p, e), t)$ where e is a global variable that indicates the length of the current string, which is now $|w| = n$. When a new character a is added after w, we can extend all open edges only with increasing the value of e by 1. Thus $e = n + 1$ now. Obviously, it only takes $O(1)$ time.
- Recall **(Case 2)** regarding the position of the active point, mentioned in Section 5.2. Assume the active point is now on the *implicit* node representing a suffix v_j of w in $CDAWG'(w)$. Then, there can be a suffix v_h such that $h \neq j$ and $v_h \equiv^R_{wa} v_j$. In such case, they become *merged* into a single explicit node $[\overset{wa}{\Longrightarrow}\overline{v_j}]^R_{wa}$, during the update of $CDAWG'(w)$ to $CDAWG'(wa)$. The equivalence test is performed on the basis of Lemma 3 to be given in the sequel.
- Consider two distinct strings $x, y \in Factor(w)$ such that $\overset{w}{\Longrightarrow}\overline{x} = x$ and $\overset{w}{\Longrightarrow}\overline{y} = y$. Assume that $x \equiv^R_w y$, that is, they are represented by the same explicit node $[x]^R_w = [y]^R_w$ in $CDAWG'(w)$. Note that, however, x and y might not be equivalent under $\equiv^R_{wa}$. This means that when $CDAWG'(w)$ is updated to $CDAWG'(wa)$, then the node has to be *separated* into two nodes $[x]^R_{wa}$ and $[y]^R_{wa}$ if $x \not\equiv^R_{wa} y$. Here we can assume $|x| > |y|$ without loss of generality. Since this node separation happens only when $x \notin Suffix(wa)$ and $y \in Suffix(wa)$, we can do this procedure after we find the end point. The condition of the node separation will be given later on, in Lemma 4.

Merging Implicit Nodes.

As mentioned above, two or more nodes implicit in $CDAWG'(w)$ can be merged into one explicit node in $CDAWG'(wa)$. For a concrete example, we show in Fig. 13 $CDAWG'(w)$ and $CDAWG'(wa)$ with $w =$ `abcabcab` and $a =$ `a`. It can be observed that the implicit nodes for `abcab`, `bcab`, and `cab` are merged into a single explicit node, and the implicit nodes for `ab` and `b` are also merged into another single explicit node in $CDAWG'(wa)$. The examination of whether to merge implicit nodes can be done by testing the equivalence

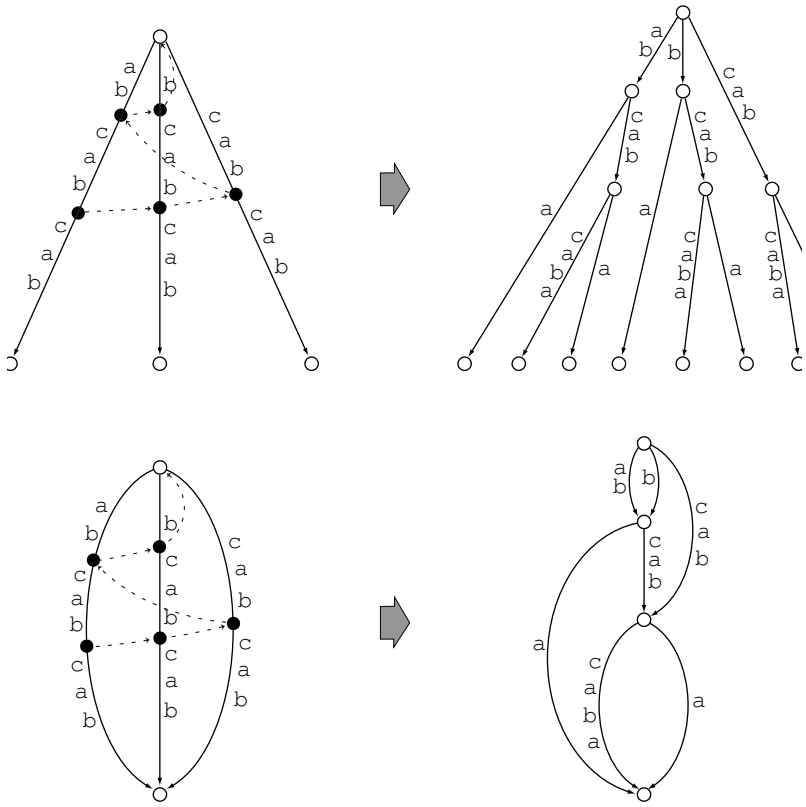

Fig. 13. Comparison of conversions. One is from $STree'(w)$ to $STree'(wa)$, while the other is from $CDAWG'(w)$ to $CDAWG'(wa)$ for $w = \texttt{abcabcab}$ and $a = \texttt{a}$. The black circles represent implicit nodes to be merged in the next step, connected by implicit suffix links corresponding to the traversal by the active point.

of two nodes under the equivalence relation $\equiv^R_{wa}$. Here, we will often use the notation of $\overrightarrow{(\cdot)}^w$ rather than $\overrightarrow{(\cdot)}^w$ in order to refer to such a node which is implicit in $CDAWG'(w)$ but becomes explicit in $CDAWG'(wa)$. The equivalence test can be performed on the basis of the following proposition and lemma.

Proposition 9 *Let $w \in \Sigma^*$. For any string $x \in Factor(w)$, let $z = \overleftrightarrow{x}^w$. Then, x occurs within z exactly once.*

Proof. By Corollary 1 we have $\overleftrightarrow{x}^w = \overleftarrow{(\overrightarrow{x}^w)}^w$. Note that there exist strings $\alpha, \beta \in \Sigma^*$ such that $\overleftarrow{(\overrightarrow{x}^w)}^w = \overleftarrow{x\alpha}^w = \beta x \alpha$. According to Proposition 2, strings α, β have to be unique, which means that x can appear in w exactly once. □

Lemma 3 *Let $w \in \Sigma^*$. For any strings $x, y \in Factor(w)$ with $y \in Suffix(x)$,*

$$x \equiv^R_w y \Leftrightarrow [\overrightarrow{x}^w]^R_w = [\overrightarrow{y}^w]^R_w.$$

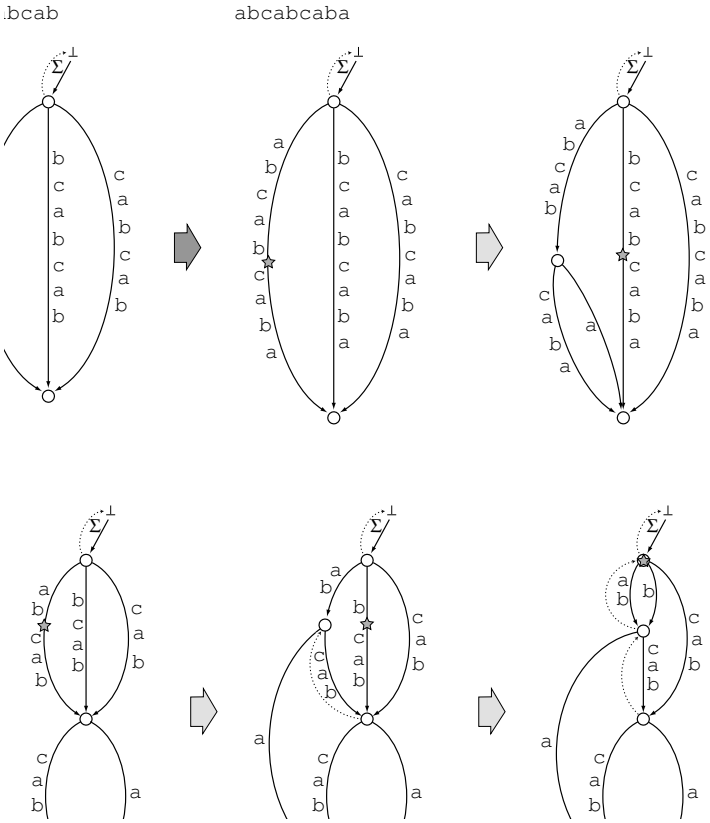

Fig. 14. Detailed conversion from $CDAWG'(w)$ to $CDAWG'(wa)$ for $w = \texttt{abcabcab}$ and $a = \texttt{a}$.

Proof. If $x \equiv_w^R y$, we have $\overleftarrow{x}^w = \overleftarrow{y}^w$ by Definition 3. By Corollary 1, we know $(\overleftarrow{x}^w)\overrightarrow{}^w = (\overrightarrow{x}^w)\overleftarrow{}^w$ and $(\overleftarrow{y}^w)\overrightarrow{}^w = (\overrightarrow{y}^w)\overleftarrow{}^w$, which yield $(\overleftarrow{x}^w)\overrightarrow{}^w\overleftarrow{}^w = (\overrightarrow{y}^w)\overleftarrow{}^w\overrightarrow{}^w$. Again by Definition 3, we have $[\overrightarrow{x}^w]_w^R = [\overrightarrow{y}^w]_w^R$.

Conversely, suppose $[\overrightarrow{x}^w]_w^R = [\overrightarrow{y}^w]_w^R$. Recall that $\overleftrightarrow{x}^w = (\overrightarrow{x}^w)\overleftarrow{}^w\overrightarrow{}^w$ by Corollary 1 and $(\overrightarrow{x}^w)\overleftarrow{}^w\overrightarrow{}^w$ is the unique longest member of $[\overrightarrow{x}^w]_w^R$. Similarly, $\overleftrightarrow{y}^w$ is the unique longest member of $[\overrightarrow{y}^w]_w^R$. Thus we have $\overleftrightarrow{x}^w = \overleftrightarrow{y}^w$. Let $z = \overleftrightarrow{x}^w = \overleftrightarrow{y}^w$. Then $z = \alpha x \beta$ for some strings α and β. Since y is a suffix of x, there exists a string δ such that $x = \delta y$. We thus have $z = \alpha\delta y\beta$. This occurrence of y in z must be the only one due to Proposition 9. Since $\overleftrightarrow{y}^w = \alpha\delta y\beta$, we conclude that every occurrence of y within w must be preceded by δ. Thus we have $x \equiv_w^R y$. □

For any string $x \in Factor(w)$, the equivalence class $[\overrightarrow{x}^w]_w^R$ is the closest explicit child of x in $CDAWG(w)$. Thus we can test the equivalence of two suffixes

x, y of w with Lemma 3.

The matter is that, for a string $x \in \mathit{Suffix}(w)$, $\overrightarrow{x}^{w}$ might not be explicit in $\mathit{CDAWG}'(w)$. Namely, on the equivalence test, we might refer to the node $[\overrightarrow{x}^{w}]^R_w$ instead of $[\overrightarrow{x}^{w}]^R_w$. Nevertheless, it does not actually happen in our on-line manner in which suffixes are processed in decreasing order of their length. See $\mathit{CDAWG}'(w)$ shown on the right of Fig. 13, where $w = \mathtt{abcabcab}$. The black points are the implicit nodes the active point traverses in the next step via 'implicit' suffix links. In $\mathit{CDAWG}'(w)$, $[\overrightarrow{\mathtt{cab}}^{w}]^R_w = [\overrightarrow{\mathtt{ab}}^{w}]^R_w = [\overrightarrow{w}^{w}]^R_w$. However, in $\mathit{CDAWG}'(wa)$, $\mathtt{cab} \not\equiv^R_{wa} \mathtt{ab}$ where $a = \mathtt{a}$. See Fig. 14 in which the detail of the update of $\mathit{CDAWG}'(w)$ to $\mathit{CDAWG}'(wa)$ is displayed. Notice that there is no trouble on merging the implicit nodes.

Separating Explicit Nodes.

When $\mathit{CDAWG}'(w)$ is updated to $\mathit{CDAWG}'(wa)$, an explicit node $[\overrightarrow{x}^{w}]^R_w$ with $x \in \mathit{Factor}(w)$ might become separated into two explicit nodes $[\overrightarrow{x}^{wa}]^R_{wa}$ and $[\overrightarrow{y}^{wa}]^R_{wa}$ if $x \notin \mathit{Suffix}(wa)$, $y \in \mathit{Suffix}(x)$, and $y \in \mathit{Suffix}(wa)$. It is inherently the same 'phenomenon' as the node separation occurring in the on-line construction of DAWGs [4]. Hereby we briefly recall the essence of the node separation of DAWGs. For $w \in \Sigma^*$ and $a \in \Sigma$, $\equiv^R_{wa}$ is a refinement of $\equiv^R_w$. Furthermore, we have the following lemma.

Lemma 4 (Blumer et al. [4]) *Let $w \in \Sigma^*$ and $a \in \Sigma$. Let $z = \mathit{LRS}(wa)$. For a string $x \in \mathit{Factor}(w)$ assume $x = \overleftarrow{x}^{w}$, that is, x is the representative of $[x]^R_w$. Then,*

$$[x]^R_w = \begin{cases} [x]^R_{wa} \cup [z]^R_{wa}, & \text{if } z \in [x]^R_w \text{ and } x \neq z; \\ [x]^R_{wa}, & \text{otherwise.} \end{cases}$$

As stated in the above lemma, we need only to care about the node $[x]^R_w$ where $z \in [x]^R_w$ and $z = \mathit{LRS}(wa)$. Namely, at most one node can be separated when a DAWG is updated with a new character. If $z \neq \overleftarrow{x}^{w}$, it is separated into two nodes $[x]^R_{wa}$ and $[z]^R_{wa}$ when $\mathit{DAWG}(w)$ is updated to $\mathit{DAWG}(wa)$ (the former case). If $z = \overleftarrow{x}^{w}$, the node is not separated (the latter case). We examine whether $z = \overleftarrow{x}^{u}$ or not by checking the length of $\overleftarrow{x}^{w}$ and z, as follows. Let $y \in \mathit{Factor}(w)$ be the string such that $\overleftarrow{y}^{w} \cdot a = z$. Note that there exists an

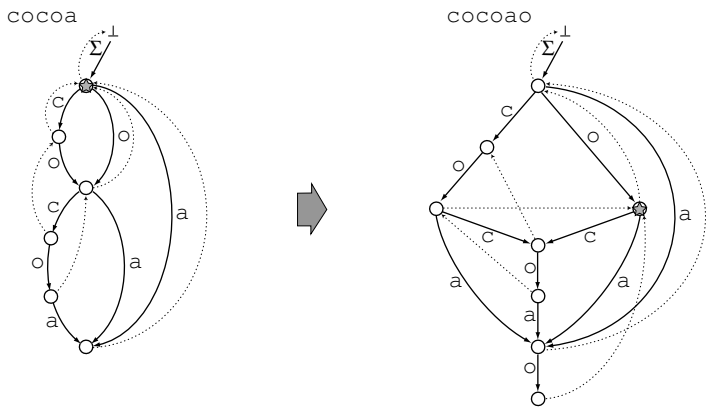

Fig. 15. Update of $DAWG(u)$ to $DAWG(ua)$, where $u = \texttt{cocoa}$ and $a = \texttt{o}$.

edge $([y]_w^R, a, [x]_w^R)$. Then,

$$z = \overleftarrow{x}^w \Leftrightarrow \mathit{length}([y]_w^R) + |a| = \mathit{length}([x]_w^R), \text{ and}$$
$$z \neq \overleftarrow{x}^w \Leftrightarrow \mathit{length}([y]_w^R) + |a| < \mathit{length}([x]_w^R).$$

If we define the length of the bottom node $\perp$ by -1, no contradiction occurs even in case that $z = \varepsilon$.

Fig. 15 shows the conversion from $DAWG(w)$ to $DAWG(wa)$ with $w = \texttt{cocoa}$ and $a = \texttt{o}$. The LRS of the string $\texttt{cocoao}$ is $\texttt{o}$, therefore we focus on edge $([\varepsilon]_w^R, \texttt{o}, [\texttt{o}]_w^R)$. Since $\mathit{length}([\varepsilon]_w^R) + |\texttt{o}| = 1 < \mathit{length}([\texttt{o}]_w^R) = 2$, node $[\texttt{o}]_w^R$ is separated into two nodes $[\texttt{co}]_{wa}^R$ and $[\texttt{o}]_{wa}^R$, as seen in $DAWG(wa)$ of Fig. 15.

Now we go back to the update of $CDAWG'(w)$ to $CDAWG'(wa)$. The test of whether to separate a node when a CDAWG is updated can also be done on the basis of Lemma 4 in a very similar way. Since only explicit nodes can be separated, we only need to care about the case that $z = \overrightarrow{\overrightarrow{z}}^{wa}$ where $z = LRS(wa)$.

Lemma 5 *Let $w \in \Sigma^*$. Let $z = LRS(w)$. Then, if $z = \overrightarrow{\overrightarrow{z}}^w$, $\overrightarrow{\overrightarrow{x}}^w = \overrightarrow{x}^w$ for any string $x \in \mathit{Factor}(w)$.*

Proof.

(1) When $x \in \mathit{Suffix}(z)$. Then we have $x \in \mathit{Suffix}(w)$ because $z = LRS(w)$. From the definition of $\overrightarrow{(\cdot)}^w$, therefore, it is obvious that $\overrightarrow{\overrightarrow{x}}^w = x$. Since $x \in \mathit{Suffix}(w)$, $\overrightarrow{x}^w = x$ from Proposition 3. Hence we have $\overrightarrow{\overrightarrow{x}}^w = \overrightarrow{x}^w$.

(2) When $x \notin \mathit{Suffix}(z)$. We here have two sub-cases to consider:

(2-a) When x is a longer suffix of w than z. Since x occurs in w exactly once and $x \equiv_w^L x\alpha$ for any non-empty string α, we have $\overrightarrow{x}^w = x$. Moreover, x is

not followed by any character in w, which means that $\overset{w}{\Longrightarrow} x = x$. Thus we have $\overset{w}{\rightarrow} x = \overset{w}{\Longrightarrow} x$.

(2-b) When x is not a suffix of w. If there exist two distinct characters a, b such that $xa, xb \in \mathit{Factor}(w)$, then $\overset{w}{\Longrightarrow} x = x$ from the definition. Moreover, $x \not\equiv_w^L xa$ since some occurrences of x in w are followed by b. Therefore we have $\overset{w}{\rightarrow} x = x$, and thus $\overset{w}{\Longrightarrow} x = \overset{w}{\rightarrow} x$.

Now assume there exists a unique non-empty string β such that $\overset{w}{\Longrightarrow} x = x\beta$. If $x\beta \in \mathit{Suffix}(w)$, then it falls into either Case (1) or (2-a). We now consider the case that $x\beta \notin \mathit{Suffix}(w)$. Since $\overset{w}{\Longrightarrow} x = x\beta$, there exist two distinct characters a, b such that $x\beta a, x\beta b \in \mathit{Factor}(w)$, and it falls into the former part of Case (2-b).

Consequently, we have $\overset{w}{\Longrightarrow} x = \overset{w}{\rightarrow} x$ in any cases. □

This lemma guarantees that the representative of $[\overset{w}{\Longrightarrow} x]_w^R$ is equal to $\overset{w}{\leftrightarrow} x$ if the preconditions in the lemma are satisfied. We can therefore execute the node separation test as follows: If $z \neq \overset{w}{\leftrightarrow} x$, it is separated into two nodes $[x]_{wa}^R$ and $[z]_{wa}^R$ when $\mathit{CDAWG}'(w)$ is updated to $\mathit{CDAWG}'(wa)$ (the former case). If $z = \overset{w}{\leftrightarrow} x$, the node $[x]_w^R$ is not separated (the latter case). We examine if $z = \overset{w}{\leftrightarrow} x$ or not by the length of $\overset{w}{\leftrightarrow} x$ and z in the following way. Let $y \in \mathit{Factor}(w)$ be the string such that $\overset{w}{\leftrightarrow} y \cdot \alpha = z$ for some string $\alpha \in \mathit{Factor}(w)$. Note that there exists an edge $([y]_w^R, \alpha, [x]_w^R)$. Then,

$$z = \overset{w}{\leftrightarrow} x \Leftrightarrow \mathit{length}([y]_w^R) + |\alpha| = \mathit{length}([x]_w^R), \text{ and}$$
$$z \neq \overset{w}{\leftrightarrow} x \Leftrightarrow \mathit{length}([y]_w^R) + |\alpha| < \mathit{length}([x]_w^R).$$

Fig. 16 shows the update of $\mathit{CDAWG}'(w)$ to $\mathit{CDAWG}'(wa)$, where $w = \texttt{cocoa}$ and $a = \texttt{o}$. The LRS of the string `cocoao` is `o`, therefore we focus on edge $([\overset{w}{\Longrightarrow} \varepsilon]_w^R, \texttt{o}, [\overset{w}{\Longrightarrow} \texttt{o}]_w^R)$. Since $\mathit{length}([\overset{w}{\Longrightarrow} \varepsilon]_w^R) + |\texttt{o}| = 1 < \mathit{length}([\overset{w}{\Longrightarrow} \texttt{o}]_w^R) = 2$, node $[\overset{w}{\Longrightarrow} \texttt{o}]_w^R$ is separated into two nodes $[\overset{wa}{\Longrightarrow} \texttt{co}]_{wa}^R$ and $[\overset{wa}{\Longrightarrow} \texttt{o}]_{wa}^R$, as seen in $\mathit{CDAWG}'(wa)$ of Fig. 16.

Pseudo-Code.

Our on-line algorithm to construct CDAWGs is described in Fig. 17 and Fig. 18. Function *extension* returns the explicit child node of a given node (implicit or explicit). Function *redirect_edge* redirects a given edge to a given node, with modifying the label of the edge accordingly. Function *split_edge* is

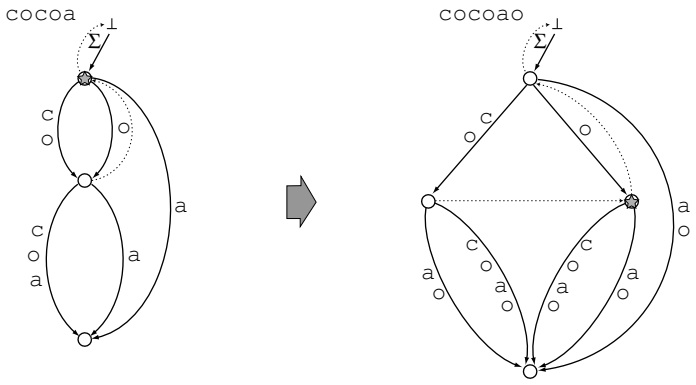

Fig. 16. Update of $CDAWG'(u)$ to $CDAWG'(ua)$, where $u = \texttt{cocoa}$ and $a = \texttt{o}$.

the same as the one used in Ukkonen's algorithm of Fig. 10, except that it also computes the length of nodes. Function *separate_node* separates a given node into two, if necessary. It is essentially the same as the separation procedure for $DAWG(w)$ given by Blumer et al. [4], except that implicit nodes are also treated.

Complexity of the Algorithm.

Theorem 6 *Assume Σ is a fixed alphabet. For any string $w \in \Sigma^*$, the proposed on-line algorithm to construct $CDAWG'(w)$ runs in $O(|w|)$ time.*

Proof. The linearity proof is in a sense the combination of that for the on-line algorithm for DAWGs [4] and that for the on-line algorithm for suffix trees [34]. We divide the time requirement into two components, both turn out to be linear. The first component consists of the total computation time by *canonize*. The second component consists of the rest.

Let $x \in Factor(w)$. We define the *suffix chain* started at x on w, denoted by $SC_w(x)$, to be the sequence of (possibly implicit) nodes reachable via suffix links from the (possibly implicit) node associated with x to the source node in $CDAWG'(w)$, as in [4]. We define its length by the number of nodes contained in the chain, and let $|SC_w(x)|$ denote it. Let k_1 be the number of iterations of the while loop of *update* and let k_2 be the number of iterations in the repeat-until loop in *separate_node*, when $CDAWG(w)$ is updated to $CDAWG(wa)$. By a similar argument as in [4], it can be derived that $|SC_{wa}(wa)| \leq |SC_w(w)| - (k_1 + k_2) + 2$. Initially $|SC_w(w)| = 1$ because $w = \varepsilon$, and then it grows at most two (possibly implicit) nodes longer in each call of *update*. Since $|SC_w(w)|$ decreases by an amount proportional to the sum of the number of iterations in the while loop and in the repeat-until loop on each call of *update*, the second time component is linear in the length of the input string.

For the analysis of the first time component we have only to consider the

```
Algorithm for on-line construction of CDAWG'(w$)
in alphabet Σ = {w[−1], w[−2], ..., w[−m]}.
/* $ is the end-marker appearing nowhere in w. */
 1 create nodes source, sink, and ⊥;
 2 for j := 1 to m do create a new edge (⊥, (−j, −j), source);
 3 suf(source) := ⊥;
 4 length(source) := 0;    length(⊥) := −1;
 5 e := 0;    length(sink) := e;
 6 (s, k) := (source, 1);    i := 0;
 7 repeat
 8     i := i + 1;    e := i;    /* e is a global variable. */
 9     (s, k) := update(s, (k, i));
10 until w[i] = $;

function update(s, (k, p)): pair of node and integers;
/* (s, (k, p − 1)) is the canonical reference pair for the active point. */
 1 c := w[p];    oldr := nil;
 2 while not check_end_point(s, (k, p − 1), c) do
 3     if k ≤ p − 1 then    /* implicit case. */
 4         if s' = extension(s, (k, p − 1)) then
 5             redirect_edge(s, (k, p − 1), r);
 6             (s, k) := canonize(suf(s), (k, p − 1));
 7             continue;
 8         else
 9             s' := extension(s, (k, p − 1));
10             r := split_edge(s, (k, p − 1));
11     else    /* explicit case. */
12         r := s;
13     create edge (r, (p, e), sink);
14     if oldr ≠ nil then suf(oldr) := r;
15     oldr := r;
16     (s, k) := canonize(suf(s), (k, p − 1));
17 if oldr ≠ nil then suf(oldr) := s;
18 return separate_node(s, (k, p));
```

Fig. 17. Main routine, **function** *update*, and **function** *check_end_point* of the on-line algorithm to construct CDAWGs.

number of iterations in the while loop in *canonize*. By considering the calls of *canonize* executed in the while loop in *update*, it results in that the total number of the iterations is linear (by the same argument as in [34]). Thus we shall consider the number of iterations of the while loop in *canonize* called in *separate_node*. There are two cases to consider:

(1) When the end point is on an implicit node. Then the computation in *canonize* takes only constant time.
(2) When the end point is on an explicit node. Let z be the LRS of w, which corresponds to the end point. Consider the last edge in the path spelling out z from the source node to the explicit node, and let the length of its label be k (≥ 1). The total number of iterations of the while loop of *canonize* in the call of *separate_node* is at most k. Since the value of k increases at most by 1 each time a new character is scanned, the time

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

function redirect_edge(s, (k, p), r);
 1 let (s, (k', p'), s') be the w[k]-edge from s;
 2 replace the edge by edge (s, (k', k' + p − k), r);

function split_edge(s, (k, p)): node;
 1 let (s, (k', p'), s') be the w[k]-edge from s;
 2 create node r;
 3 replace the edge by (s, (k', k' + p − k), r) and (r, (k' + p − k + 1, p'), s');
 4 length(r) := length(s) + (p − k + 1);
 5 return r;

function separate_node(s, (k, p)): pair of node and integer;
 1 (s', k') := canonize(s, (k, p));
 2 if k' ≤ p then return (s', k');    /* implicit case. */
 3 /* explicit case. */
 4 if length(s') = length(s) + (p − k + 1) then return (s', k');
 5 create node r' as a duplication of s' with the out-going edges;
 6 suf(r') := suf(s');    suf(s') := r';
 7 length(r') := length(s) + (p − k + 1);
 8 repeat
 9     replace the w[k]-edge from s to s' by edge (s, (k, p), r');
10     (s, k) := canonize(suf(s), (k, p − 1));
11 until (s', k') ≠ canonize(s, (k, p));
12 return (r', p + 1);
```

Fig. 18. Other functions for the on-line algorithm to construct CDAWGs. Since **function** *check_end_point* and **function** *canonize* used here are identical to those shown in Fig. 10, they are omitted.

requirement of the while loop of *canonize* in *separate_node* is bounded by the total length of the input string.

As a result of the above discussion, we can finally conclude that the first and second components take overall linear time. □

7 Construction of CDAWGs for a Set of Strings

In the previous chapters we discussed on-line construction of index structures for a single string $w \in \Sigma^*$. On the other hand, we now consider such case that we are given a set S of strings as an input. The suffix trie, suffix tree, DAWG, and CDAWG for S can all be well-defined. Any index structure for S must represent all strings in $Factor(S)$.

Blumer et al. [5] introduced DAWGs for a set of strings, and presented an algo-

rithm that builds $DAWG(S)$ in $O(\|S\|)$ time. They also introduced CDAWGs for a set of strings. Their algorithm for construction of $CDAWG(S)$ runs in linear time in the input size (namely, in $O(\|S\|)$ time), but not in time linear in the output size because it first builds $DAWG(S)$ and then converts it to $CDAWG(S)$ by deleting internal nodes of out-degree one and concatenating their edges accordingly. Kosaraju [26] introduced the suffix tree for a set S of strings, often referred to the *generalized suffix tree* for S. A slight modification of Ukkonen's algorithm recalled in Section 5.2 is capable of constructing $STree'(S)$ in $O(\|S\|)$ time.

In this section, we give the first algorithm which builds $CDAWG'(S)$ in $O(\|S\|)$ time, and directly. Let $S = \{w_1, w_2, \ldots, w_k\}$, where $k = |S|$. Then we consider set $S' = \{w_i\$_i \mid w_i \in S \text{ and } \$_i \notin Factor(S) \text{ for any } 1 \leq i \leq |S|\}$. Notice that S' has the prefix property, and thus, $CDAWG'(S') = CDAWG(S')$ for any S. $CDAWG'(S')$ can be constructed by a slight modification of the algorithm proposed in the previous section. We use a global variable e_i for each string in S', where $1 \leq i \leq |S|$, which indicates the ending position of the open edges for each string. The set S' is input in the form of a single stream $s = w_1\$_1 w_2\$_2 \cdots w_k\$_k$. Whenever we encounter an end-marker $\$_i$ in the input, we stop increasing the value of e_i. Then we create the new $(i+1)$-th sink node, and start increasing the value of e_{i+1} each time a new character is scanned until encountering $\$_{i+1}$. Hereby we have the following:

Theorem 7 *Assume Σ is a fixed alphabet. For any set S of strings, the proposed algorithm directly constructs $CDAWG'(S')$ on-line and in $O(\|S'\|)$ time, using $O(\|S'\|)$ space.*

Fig. 19 shows construction of $CDAWG'(S')$, where $S' = \{\texttt{cocoa}\$_1, \texttt{cola}\$_2\}$.

8 Conclusion

The *compact directed acyclic word graph* (*CDAWG*) of a string w is a space-economical index structure that represents all factors of w. CDAWGs have so far been studied in literature such as [5,13].

In this paper we presented an on-line linear-time algorithm for constructing CDAWGs for a single string. The algorithm can easily be extended to construction of CDAWGs for a set of strings, running in linear time with respect to the total length of the strings in the set. A CDAWG can be obtained by compacting the corresponding DAWG or minimizing the corresponding suffix tree; however, our approach permits us to save time and space simultaneously, since the CDAWG can be built directly. Moreover, in our on-line manner the CDAWG for wa can be built simply by updating the CDAWG of w with the

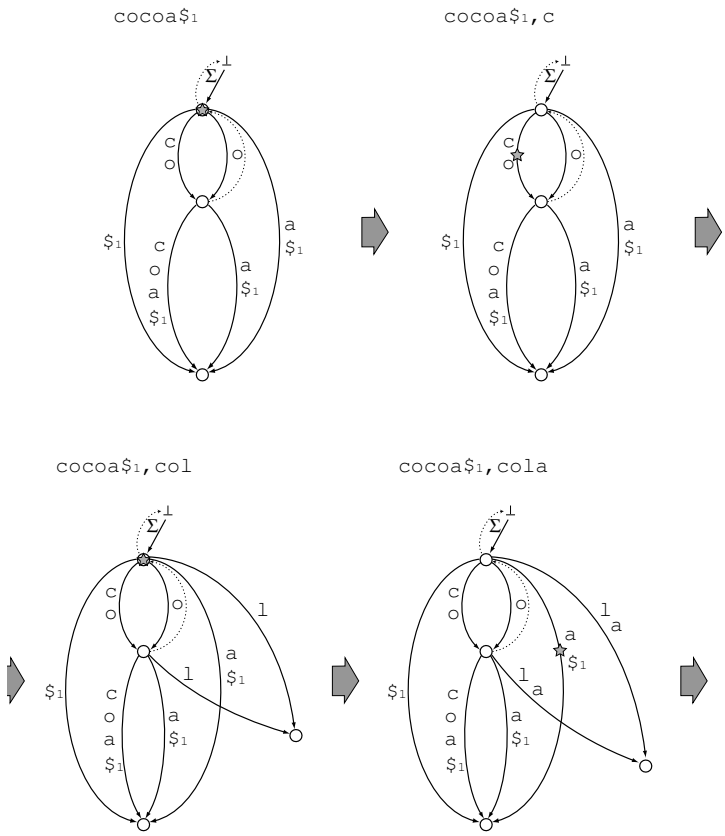

Fig. 19. Construction of $CDAWG'(S')$ for $S' = \{\texttt{cocoa\$}_1, \texttt{cola\$}_2\}$.

new character a, in contrast with Crochemore and Vérin's off-line algorithm in [13] which needs to reconstruct it from scratch.

There have been made further work on CDAWGs. In [20,22] a *generic* algorithm to construct suffix tries, suffix trees, DAWGs, and CDAWGs was considered. The CDAWG of a *trie* was introduced in [19], where the trie is a tree structure representing a set of strings. An algorithm to construct the CDAWG for a given trie was also given in that paper, which runs in time proportional to the number of nodes in the trie. In [21], an on-line algorithm that builds *symmetric CDAWGs* (*SCDAWGs*) was introduced. The SCDAWG of a string w was first introduced in [5], which represents all factors of both w and its reversed string. Moreover, a liner-time algorithm to construct and maintain CDAWGs for a *sliding window* was given in [24]. An application of CDAWGs for a sliding window is the *prediction by partial matching* (*PPM*) style statistical data compression model [8,7].

Not only are CDAWGs attractive as an index structure, but also the underlying equivalence relation is useful in data mining and machine discovery from textual databases. In fact, the equivalence relation has played a central role in supporting expert researchers working on evaluating and interpreting literary knowledge mined from anthologies of classical Japanese poems [32].

References

[1] A. Apostolico. The myriad virtues of subword trees. In *Combinatorial Algorithm on Words*, volume 12 of *NATO Advanced Science Institutes, Series F*, pages 85–96. Springer-Verlag, 1985.

[2] A. Apostolico and S. Lonardi. A speed-up for the commute between subword trees and DAWGs. *Information Processing Letters*, 83(3):159–161, 2002.

[3] M. Balík. Implementation of DAWG. In *Proc. The Prague Stringology Club Workshop '98 (PSCW'98)*, pages 26–35. Czech Technical University, 1998.

[4] 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*, 40:31–55, 1985.

[5] 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.

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

[7] J. G. Cleary, W. J. Teahan, and I. H. Witten. Unbounded length contexts for PPM. In *Proc. Data Compression Conference '95 (DCC'95)*, pages 52–61. IEEE Computer Society, 1995.

[8] J. G. Cleary and I. H. Witten. Data compression using adaptive coding and partial string matching. *IEEE Trans. Commun.*, 32(4):396–402, 1984.

[9] M. Crochemore. Transducers and repetitions. *Theoretical Computer Science*, 45:63–86, 1986.

[10] M. Crochemore. Reducing space for index implementation. *Theoretical Computer Science*, 292(1):185–197, 2003.

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

[12] M. Crochemore and W. Rytter. *Jewels of Stringology*. World Scientific, 2002.

[13] M. Crochemore and R. Vérin. On compact directed acyclic word graphs. In *Structures in Logic and Computer Science*, volume 1261 of *Lecture Notes in Computer Science*, pages 192–211. Springer-Verlag, 1997.

[14] G. Gonnet, R. Baeza-Yates, and T. Snider. New indices for text: PAT trees and PAT arrays. *Information retrieval: data structures and algorithms*, pages 66–82, 1992.

[15] R. Grossi and J. S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In *Proc. of 32nd ACM Symposium on Theory of Computing (STOC'00)*, pages 397–406, 2000.

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

[17] J. Holub and B. Melichar. Approximate string matching using factor automata. *Theoretical Computer Science*, 249:305–311, 2000.

[18] S. Inenaga. Bidirectional construction of suffix trees. *Nordic Journal of Computing*, 10(1):52–67, 2003.

[19] S. Inenaga, H. Hoshino, A. Shinohara, M. Takeda, and S. Arikawa. Construction of the CDAWG for a trie. In *Proc. The Prague Stringology Conference '01 (PSC'01)*, pages 37–48. Czech Technical University, 2001.

[20] S. Inenaga, H. Hoshino, A. Shinohara, M. Takeda, and S. Arikawa. On-line construction of compact directed acyclic word graphs. Technical Report DOI-TR-CS-183, Department of Informatics, Kyushu University, 2001.

[21] S. Inenaga, H. Hoshino, A. Shinohara, M. Takeda, and S. Arikawa. On-line construction of symmetric compact directed acyclic word graphs. In *Proc. of 8th International Symposium on String Processing and Information Retrieval (SPIRE'01)*, pages 96–110. IEEE Computer Society, 2001.

[22] S. Inenaga, H. Hoshino, A. Shinohara, M. Takeda, and S. Arikawa. Unification of algorithms to construct index structures for texts. Technical Report DOI-TR-CS-196, Department of Informatics, Kyushu University, 2001.

[23] 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 Annual Symposium on Combinatorial Pattern Matching (CPM'01)*, volume 2089 of *Lecture Notes in Computer Science*, pages 169–180. Springer-Verlag, 2001.

[24] S. Inenaga, A. Shinohara, M. Takeda, and S. Arikawa. Compact directed acyclic word graphs for a sliding window. In *Proc. 9th International Symposium on String Processing and Information Retrieval (SPIRE'02)*, volume 2476 of *Lecture Notes in Computer Science*, pages 310–324. Springer-Verlag, 2002.

[25] J. Kärkkäinen. Suffix cactus: A cross between suffix tree and suffix array. In *Proc. 6th Annual Symposium on Combinatorial Pattern Matching (CPM'95)*, volume 973 of *Lecture Notes in Computer Science*, pages 191–204. Springer-Verlag, 1995.

[26] S. R. Kosaraju. Fast pattern matching in trees. In *Proc. 30th IEEE Symp. on Foundations of Computer Science*, pages 178–183, 1989.

[27] S. Kurtz. Reducing the space requirement of suffix trees. *Software - Practice and Experience*, 29(13):1149–1171, 1999.

[28] V. Mäkinen. Compact suffix array - a space-efficient full-text index. *Fundamenta Infomaticae*, 56(1-2):191–210, 2003.

[29] U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

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

[31] K. Sadakane. Compressed text databases with efficient query algorithms based on the compressed suffix array. In *Proc. of 11th International Symposium on Algorithms and Computation (ISAAC'00)*, volume 1969 of *Lecture Notes in Computer Science*, pages 410–421. Springer-Verlag, 2000.

[32] M. Takeda, T. Matsumoto, T. Fukuda, and I. Nanri. Discovering characteristic expressions from literary works. *Theoretical Computer Science*, 292(2):525–546, 2003.

[33] E. Ukkonen. Approximate string matching over suffix trees. In *Proc. 4th Annual Symposium on Combinatorial Pattern Matching (CPM'93)*, volume 684 of *Lecture Notes in Computer Science*, pages 228–242. Springer-Verlag, 1993.

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

[35] E. Ukkonen and D. Wood. Approximate string matching with suffix automata. *Algorithmica*, 10(5):353–364, 1993.

[36] P. Weiner. Linear pattern matching algorithms. In *Proc. 14th Annual Symposium on Switching and Automata Theory*, pages 1–11, 1973.