On Data Structures for String Searching Problems

Shunsuke Inenaga

February, 2002

Department of Informatics
Kyushu University

Contents

Chapter 1

Introduction

1.1 A Research Area Called 'Stringology'

In Theoretical Computer Science, sequences of characters are often referred to as *strings*. The design of algorithms that processes and manipulates strings is one of the most active subfield of general algorithmic research. *Stringology* has been the nickname of this new area [14, 3]. It covers any, and various, kinds of problems related to strings and things that can be regarded as or transformed into strings such as music data [13, 34], DNA and protein data [20].

Natural language texts and biological sequences, like those electrically available via WWW or from biological databases, can be arose as typical examples of vast strings frequently utilized today. Since a huge, and growing, amount of string data is at present stored and requires processing, once acceptable naive algorithms are no longer effective. In the present information age, it directly causes neccessity to develop innovative string processing algorithms, which must be correct and efficient both in time and space. That is the reason why Stringology is one particular area that has been extremely active for the last few decades.

String processing problems include a variaty of applications such as exact pattern matching, approximate pattern matching, similarity measurement between two strings, locating repetition in strings, string compression, pattern matching on compressed strings, two-dimensional pattern matching, and so on. Among those, the most fundamental and important problem is the first one, *exact pattern matching on strings*.

The exact pattern matching problem is formalized as follows: Given a text string w

and a pattern string p, return "yes" if p occurs in w, and "no" otherwise. There are two situations for the problem, namely, one that w is static and p is dynamic, or vice versa. In the former case, it is appropriate to construct a data structure for w that serves as indexes of w. Such a structure is called an *index structure* for the text string w. Let n and m be the length of w and p, respectively. An index structure for w must support all *factors* of w, where a factor of w is a string occuring within w. Unfortunately, the number of factors of w is $O(n^2)$, although it is favorable to implement the structure with as little memory space as possible, desirably with $O(n)$ space.

1.2 Linear-Space Index Structures

In 1973, there was a 'big bang' in Stringology due to Weiner [58]. He succeeded to introduce an index structure called a *suffix tree*, which only requires $O(n)$ space. He furthermore developed an algorithm to construct the suffix tree for a string w in $O(n)$ time. By means of the suffix tree of w, it is possible to solve the pattern matching problem in $O(n)$ preprocessing time and $O(m)$ searching time. This result is very surprising and its contribution is immeasurable. In fact, it is claimed that Knuth later on referred to Weiner's algorithm as 'the algorithm of 1973' [2]. Suffix trees are useful not only for the above simple pattern matching problem but also for a 'myriad' [2] of other applications represented by the problem to find the longest common factors of two strings in linear time, and so on. It is claimed that Knuth had conjectured in 1970 that linear time solution to this problem was impossible to achieve [36]. This fact also tells us that the invention of suffix trees was truly revolutionary.

Following Weiner's invention, McCreight in 1976 proposed a more space-economical algorithm for the construction of suffix trees [43]. It has permitted us to save more memory space on building suffix trees (in constant term).

However, both algorithms above mentioned have been thought to be considerably complicated. It might have caused the delay of the spread of suffix trees, in spite of their remarkable usefulness. More recently, this matter was settled in 1995 by Ukkonen's suffix tree construction algorithm [55]. Ukkonen's algorithm is believed to be the conceptually easiest to understand, and elegant [20, 18]. It has a certain helpful property called *on-line*, i.e., it processes a given string from left to right, one by one, while constructing the suffix tree for the string already scanned at each step, with no need to read the whole string beforehand. In addition, it allows us to construct the suffix tree for a set of strings in time linear in the total length of the strings.

On the other hand, in 1985, Blumer et al. introduced counterpart of suffix trees, called *directed acyclic word graphs* (*DAWGs*) [7]. The DAWG of a string w is known to be the smallest automaton that accepts all suffixes of w [12]. One drawback of the suffix tree for a string w is that the label of each edge needs to be implemented by a pair of integers which respectively represent the beginning and ending positions of the label string in w. It means that we have to store the input string w in order to keep the suffix tree with linear space. Conversely, the label of any edge of the DAWG for w consists of a character, not a string. Since there is no need to keep the input string w if the DAWG for w is once completed, we can then delete w from main memory.

An algorithm to construct the DAWG of a given string was also proposed by Blumer et al., which processes the string on-line, and runs in linear time [7]. In 1987 Blumer et al. moreover gave an on-line algorithm that constructs the DAWG for a given set of strings, running in time linear in the total length of strings [8].

A lot more space-economical index structure is a *compact directed acyclic word graph* (*CDAWG*), introduced by Blumer et al. as well [8]. It has been shown that CDAWGs require strictly smaller space than suffix trees and DAWGs, both theoretically and experimentally. A typical biological sequence may be many millions of characters long. Thus it is quite significant to reduce the constant term usually ignored due to the big-O notation for the space and time complexity.

Blumer et al. gave an algorithm that constructs the CDAWG for a string w, by once building the DAWG of w and then shrinking it into the corresponding CDAWG. The CDAWG for a set of strings can also be built similarly. The drawback of this method is that we have to construct the corresponding DAWG as an intermediate, which contains many nodes and edges turning out to be redundant in the CDAWG. It was 1997 when the first algorithm to *directly* construct the CDAWG of a given string was developed by Crochemore and Vérin [16]. Their algorithm is based on McCreight's suffix tree construction algorithm. Thereby, their algorithm does not have the on-line property which can be useful in some situations.

All the structures mentioned above are automata-oriented. Another idea to represent all factors of a given string in a data structure is based on arrays. The first array of this kind is the *suffix array* [41], followed by the *suffix cactus* [35], the *compact suffix array* [40], and the *compressed suffix array* [19, 47]. They are in general more space-economical than those automata-oriented structures (in constant term), but they instead sacrifice searching time. Namely, searching p in w by using an array takes $O(m + \log n)$ time.

1.3 Our Contribution

In the following chapters, we report our contribution to Stringology. We have chosen the automata-oriented index structures to work for, since we wish the situation where we can find the occurrence of a pattern p in a text w *as fast as possible*, in $O(m)$ searching time.

In Chapter 2.3, we define index structures basing on the equivalence classes on strings. They are defined in common notations, thus give us a good 'unified' view on the structures which reveals their insights and relationship among them.

In Chapter 3, we focus our attention on CDAWGs. An algorithm which constructs CDAWGs in on-line manner had been a long-term missing piece, as remarked in Section 1.2. In the chapter, we report the success of invention of an on-line algorithm that constructs the CDAWG for a given string. We prove that the algorithm proposed runs in linear time. Also, we show that the CDAWG for a set of strings can also be built by a straightforward extension of the algorithm.

We dedicate Chapter 4 to the introduction of more surprising fact about the unified view for the index structures. To be concrete, we introduce a *generic* algorithm that is capable of constructing any of suffix tries, suffix trees, DAWGs, and CDAWGs. This saves us a great deal of effort to implement distinct algorithms for all the index structures. In addition, it gives us insight into their properties, what is common to and what is different amongst them, from algorithmic point of view.

Chapter 5 is devoted to a report of an algorithm for the construction of the CDAWG for a given trie that 'compactly' represents a set of strings. Since the trie for a set of strings shares their common prefixes, the number of nodes in it is generally smaller than the total length of the strings. The algorithm is a non-trivial extension of the one given in Chapter 3. We establish that the algorithm performs in time linear in the number of nodes in a given trie.

In Chapter6, we consider not only an index structure for a given string w, but also that for the reversal of w, denoted by w^{rev}. It is a well known property that the suffix tree of w and the DAWG for w^{rev} can share the same nodes [11]. Therefore, they can be represented together with, and can be seen as one structure. In the chapter, we introduce an on-line algorithm which simultaneously constructs both the suffix tree of w and the DAWG for w^{rev}. Furthermore, the chapter is dedicated to the *symmetric compact directed acyclic word graph* (*SCDAWG*) of w, which also supports both indexes of w and w^{rev} [7]. We report that we developed an on-line algorithm which builds the SCDAWG for a given string w in $O(n)$ time.

In Chapter 7, a *bidirectional* on-line linear-time construction algorithm for suffix trees is presented. As mentioned in Section 1.2, Ukkonen's algorithm allows us to update an input string by adding new strings at its right. Nevertheless, we cannot extend an input string to the left direction in using his algorithm. The result to be reported in this chapter means that we can extend an input string to both direction, without re-constructing the suffix tree from scratch. Furthermore, we show the DAWG for w can also be constructed in bidirectional on-line manner, and in linear time.

In Chapter 8, we consider the collection of DAWGs for all suffixes of a given string w. It is called the *naive all-suffixes directed acyclic word graph* (*naive ASDAWG*) for w. It is clear that the size of the naive ASDAWG for w is $O(n^2)$. We report that we succeeded to develop a new structure, named the *minimum all-suffixes directed acyclic word graph* (*MASDAWG*). The MASDAWG of w is the minimized version of the naive ASDAWG for w. We prove its size is $\Theta(n)$ if the alphabet Σ is unary, and $\Theta(n^2)$ otherwise. An on-line algorithm that directly constructs the MASDAWG for w in time proportional to its size is also given.

Given two sets of strings, it is a quite important problem in Knowledge Discovery and Data Mining to find a *rule* which separates them. The accuracy of the separation and the simplicity of the rule depend on what sort of pattern we adopt. There had been algorithms in which *substring* patterns [48] and *subsequence* patterns [23] are utilized in order to distinguish two given sets of strings. In Chapter 9, we report the work where we extended the algorithms by applying *episode* patterns as rules. We succeeded to develop a practical, efficient algorithm to find the best episode patterns to separate two sets of strings. In [27], it is experimentally shown that the result of our work is superior to its previous versions. Also, we in this chapter emphasize that MASDAWGs, introduced in Chapter 8, are believed to be powerful 'weapon' to propose a new practical algorithm to find a *variable-length-don't-care's* pattern (*VLDC*-pattern) that efficiently separates given two sets of strings.

Chapter 2

Preliminaries

2.0.1 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 $\Sigma^+ = \Sigma^* - \{\varepsilon\}$. Let y be a string such that $y = b_1 b_2 \cdots b_m$ where $m \geq 1$ and $b_j \in \Sigma$ for $1 \leq j \leq m$. Then, string $a_1 a_2 \cdots a_n b_1 b_2 \cdots b_m$ is said to be the *concatenation* of x and y, and denoted by $x \cdot y$, or simply, by xy. For any string $x \in \Sigma^*$,

$$x\varepsilon = \varepsilon x = x.$$

Strings x, y, and z are said to be a *prefix*, *factor*, and *suffix* of string $w = xyz$, respectively. The sets of 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, that is,

$$\|S\| = \sum_{k=1}^{\ell} |w_k|.$$

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 2.1 *Let $S = \{w_1, \dots, w_k\}$ where $w_i \in \Sigma^*$ for $1 \le i \le k$ and $k \ge 1$. We say that S has the* prefix property *iff $w_i \notin \mathit{Prefix}(w_j)$ for any $1 \le i \ne j \le k$.*

2.0.2 Equivalence Relations on Strings

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

Definition 2.2 *Let $S \subseteq \Sigma^*$. The equivalence relations $\equiv^L_S$ and $\equiv^R_S$ on Σ^* are defined by*

$$\begin{aligned} x \equiv^L_S y &\Leftrightarrow \mathit{Prefix}(S)x^{-1} = \mathit{Prefix}(S)y^{-1}, \\ x \equiv^R_S y &\Leftrightarrow x^{-1}\mathit{Suffix}(S) = y^{-1}\mathit{Suffix}(S). \end{aligned}$$

The equivalence class of a string $x \in \Sigma^*$ with respect to $\equiv^L_S$ (resp. $\equiv^R_S$) is denoted by $[x]^L_S$ (resp. $[x]^R_S$).

If $S = \{\mathtt{cocoa}, \mathtt{cola}\}$, $[\mathtt{c}]^L_S = \{\mathtt{c}, \mathtt{co}\}$, $[\mathtt{o}]^L_S = \{\mathtt{o}\}$, $[\mathtt{l}]^L_S = \{\mathtt{l}, \mathtt{la}\}$, $[\mathtt{c}]^R_S = \{\mathtt{c}\}$, $[\mathtt{o}]^R_S = \{\mathtt{o}, \mathtt{co}\}$, $[\mathtt{l}]^R_S = \{\mathtt{l}, \mathtt{ol}, \mathtt{col}\}$, and so on.

Note that all strings that are not in $\mathit{Factor}(S)$ form one equivalence class under $\equiv^L_S$. This equivalence class is called the *degenerate* class. All other classes are called *non-degenerate*. It follows from the definition of $\equiv^L_S$ that, if two strings $x, y \in \mathit{Factor}(S)$ are in the same equivalence class under $\equiv^L_S$, then either x is a prefix of y, or vice versa. Therefore, each equivalence class in $\equiv^L_S$ other than the degenerate class has a unique longest member. A similar argument holds for $\equiv^R_S$.

Definition 2.3 *For any string $x \in \mathit{Factor}(S)$, $\overset{S}{\overrightarrow{x}}$ (resp. $\overset{S}{\overleftarrow{x}}$) denotes the unique longest member of $[x]^L_S$ (resp. $[x]^R_S$). We call $\overset{S}{\overrightarrow{x}}$ (resp. $\overset{S}{\overleftarrow{x}}$) the* representative *of $[x]^L_S$ (resp. $[x]^R_S$).*

For any string $x \in \mathit{Factor}(S)$, there uniquely exist strings α and β such that $\overset{S}{\overleftarrow{x}} = \alpha x$ and $\overset{S}{\overrightarrow{x}} = x\beta$. In the running example, $\overset{S}{\overrightarrow{\mathtt{c}}} = \mathtt{co}$, $\overset{S}{\overrightarrow{\mathtt{o}}} = \mathtt{o}$, $\overset{S}{\overrightarrow{\mathtt{l}}} = \mathtt{la}$, $\overset{S}{\overleftarrow{\mathtt{c}}} = \mathtt{c}$, $\overset{S}{\overleftarrow{\mathtt{o}}} = \mathtt{co}$, $\overset{S}{\overleftarrow{\mathtt{l}}} = \mathtt{col}$.

Definition 2.4 *For any string $x \in \mathit{Factor}(S)$, let $\overset{S}{\overleftrightarrow{x}}$ be the string $\alpha x \beta$ ($\alpha, \beta \in \Sigma^*$) such that $\overset{S}{\overleftarrow{x}} = \alpha x$ and $\overset{S}{\overrightarrow{x}} = x\beta$.*

What $\overset{S}{\overleftrightarrow{x}} = \alpha x \beta$ implies is that:

(1) Every time x occurs in $w \in S$, it is preceded by α and followed by β within w.

(2) α and β are the longest strings satisfying (1).

In the running example, $\overset{S}{\overleftrightarrow{\texttt{c}}} = \texttt{co}$ and $\overset{S}{\overleftrightarrow{\texttt{o}}} = \texttt{co}$, $\overset{S}{\overleftrightarrow{\texttt{l}}} = \texttt{cola}$.

Definition 2.5 *Let x, y be arbitrary strings in Σ^*. We write $x \equiv_S y$ if,*

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

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

For any string $x \in Factor(S)$, $\overset{S}{\overleftrightarrow{x}}$ is the unique longest member of $[x]_S$, and is called the representative of $[x]_S$.

Lemma 2.1 (Blumer et al. [8]) *The equivalence relation $\equiv_S$ is the transitive closure of the relation $\equiv_S^L \cup \equiv_S^R$.*

It follows from the lemma above that

Corollary 2.1 *For any string $x \in Factor(S)$,*

$$\overset{S}{\overleftrightarrow{x}} = \overset{S}{\overrightarrow{(\overset{S}{\overleftarrow{x}})}} = \overset{S}{\overleftarrow{(\overset{S}{\overrightarrow{x}})}}.$$

The number of the strings in $Factor(S)$ is $O(\|S\|^2)$. However, the number of strings x such that $x = \overset{S}{\overrightarrow{x}}$ (or $\overset{S}{\overleftarrow{x}}$) is $O(\|S\|)$. The following lemma gives tighter upper bounds.

Lemma 2.2 (Blumer et al. [8]) *Assume that $\|S\| > 1$. The number of the non-degenerate equivalence classes in $\equiv_S^L$ (or $\equiv_S^R$) is at most $2\|S\| - 1$. The number of the non-degenerate equivalence classes in $\equiv_S$ is at most $\|S\| + |S|$.*

If S is a singleton $\{w\}$ where $w \in \Sigma^*$, throughout this paper the notations defined for S are written by using w instead of S, as,

$$\equiv_w^L, \equiv_w^R, \equiv_w, [(\cdot)]_w^L, [(\cdot)]_w^R, [(\cdot)]_w, \overset{w}{\overrightarrow{(\cdot)}}, \overset{w}{\overleftarrow{(\cdot)}}, \overset{w}{\overleftrightarrow{(\cdot)}}.$$

2.1 Graphs and Trees

Let V be a finite set of *nodes*. Let E be a finite set of *edges*, namely, that of pairs of nodes. Then $G = (V, E)$ is said to be a *directed graph*.

In a directed graph $G = (E, V)$, the sequence of nodes $u_0, u_1, \ldots, u_n$ is called a *path* if $(u_{i-1}, u_i) \in E$ for each i $(1 \leq i \leq n)$. The *depth* of the path is n. A path with $u_0 = u_n$ is called a *cycle*. If G has no cycle, it is called a *directed acyclic graph* (*DAG* for short).

An edge (u, v) is said to be an *out-going* edge of u and an *in-coming* edge of v. The number of in-coming (resp. out-going) edges of a node u is said to be the *in-degree* (resp. *out-degree*) of u.

A directed graph T with the following properties is called a *tree*.

- There uniquely exists a node of in-degree 0 in T. It is called the *root* node.

- For any node u in T, there uniquely exists a path from the root node to u.

If $(u, v) \in T$, then u is said to be a *parent* node of v, and v is said to be a *child* node of u. Any node in a tree other than the root node has its unique parent node. A node of out-degree zero is called a *leaf* node. A node that is neither the root node nor a leaf node is called an *internal* node. If there is a path from a node u to a node v, u is said to be an *ancestor* of v, and v is said to be a *descendant* of u.

2.1.1 Tries

We here consider an edge-labeled tree $T = (V, E)$ with $E \subseteq V \times \Sigma^+ \times V$ where the second component of each edge represents its label. Let S be a set of strings. The tree representing all strings in S is called the *trie* and denoted by $\mathit{Trie}(S)$.

Definition 2.6 *$\mathit{Trie}(S)$ is the tree (V, E) such that*

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

If S has the prefix property, each string in S is represented by a leaf node in $\mathit{Trie}(S)$. It is sometimes favorable if all strings are associated with leaf nodes. In such case, we consider the set S' such that

$$S' = \{w_i\$_i \mid w_i \in S \text{ and } \$_i \notin \Sigma \text{ for } 1 \leq i \leq |S|\}.$$

For any set S of strings, S' has the prefix property. Hence every string in S' is represented by a lead node in $\mathit{Trie}(S')$.

2.2 Deterministic Finite State Automata

A *deterministic finite automaton* (*DFA* for short) is a quintuplet $M = (Q, \Sigma, \delta, q_0, F)$, where;

Q is a non-empty set. Its elements are called *states*.

Σ is an alphabet.

δ is a function $Q \times \Sigma \to Q$. It is called the *state-transition function*.

$q_0 \in Q$ is the *initial* state.

F is a subset of Q. Its elements are called *accepting* states.

We extend the state-transition function $\delta : Q \times \Sigma \to Q$ to $\hat{\delta} : Q \times \Sigma^* \to Q$, as follows.

$$\begin{cases} \hat{\delta}(q, \varepsilon) & = \quad q \quad (q \in Q) \\ \hat{\delta}(q, xa) & = \quad \delta(\hat{\delta}(q, x), a) \quad (q \in Q, a \in \Sigma, x \in \Sigma^*) \end{cases}$$

Let w be an arbitrary string in Σ^*. If $\hat{\delta}(q_0, w) \in F$, we say that w is *accepted* by DFA M. We can examine in $O(|w|)$ time whether or not w is accepted by DFA M.

2.3 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 set S of strings, denoted by $\mathit{STrie}(S)$, $\mathit{STree}(S)$, $\mathit{DAWG}(S)$, and $\mathit{CDAWG}(S)$, respectively. All these structures represent every string $x \in \mathit{Factor}(S)$. 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 [58, 43, 55, 7, 8, 16].

2.3.1 Suffix Tries

Definition 2.7 *$STrie(S)$ is the tree (V, E) such that*

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

and its suffix links are the set

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

Each string $x \in Factor(S)$ has a one-to-one correspondence to a certain node in $STrie(S)$. The root node of $STrie(S)$ corresponds to ε. If $Suffix(S) - \{\varepsilon\}$ has the prefix property, every string in $Suffix(S) - \{\varepsilon\}$ is represented by a leaf node in $STrie(S)$.

If $|S| = 1$, $STrie(S)$ is also written as $STrie(w)$ where $S = \{w\}$. $STrie(\texttt{coco})$ and $STrie(\texttt{cocoa})$ are displayed in Figure 2.1 together with their suffix links.

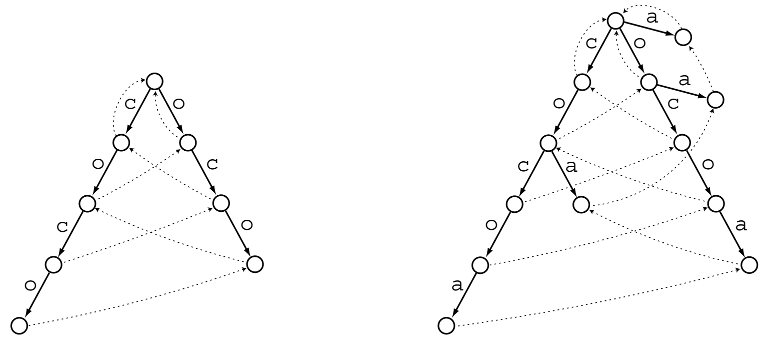


Figure 2.1: $STrie(\texttt{coco})$ on the left, and $STrie(\texttt{cocoa})$ on the right. The solid arrows represent the edges, while the dotted arrows denote the suffix links.

2.3.2 Suffix Trees

Definition 2.8 *$STree(S)$ is the tree (V, E) such that*

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

and its suffix links are the set

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

The root node of $STree(S)$ is associated with $\overset{S}{\overrightarrow{\varepsilon}}$. If $Suffix(S) - \{\varepsilon\}$ has the prefix property, every string in $Suffix(S) - \{\varepsilon\}$ is represented by a leaf node in $STree(S)$. If $|S| = 1$, $STree(S)$ is also written as $STree(w)$ where $S = \{w\}$.

The node set of $STree(S)$ is a subset of that of $STrie(S)$, as seen in the definitions. It means that a string in $Factor(S)$ might be represented on an edge in $STree(S)$. In this case, we say that the string is represented in an *implicit* node. Conversely, every string in the node set V of $STree(S)$ is said to be represented in an *explicit* node. For example, in $STree(\texttt{coco})$ of Figure 2.2, string `c` is represented by an implicit node, while string `co` is on an explicit node.

$STree(S)$ can be seen as the compacted version of $STrie(S)$ with "$\overset{S}{\overrightarrow{(\cdot)}}$ operation". See $STrie(\texttt{cocoa})$ in Figure 2.1 and $STree(\texttt{cocoa})$ in Figure 2.2. $STree(\texttt{cocoa})$ can be obtained by removing any internal nodes of out-degree one in $STrie(\texttt{cocoa})$, and suffix links associated with the removed nodes are also deleted. However, this approach cannot derive $STree(\texttt{coco})$ from $STrie(\texttt{coco})$ (see Figure 2.1 and Figure 2.2). That is, even if a node $\overset{S}{\overrightarrow{x}}$ is of out-degree one in $STrie(S)$, it is not removed if $\overset{S}{\overrightarrow{x}} \in Suffix(S)$.

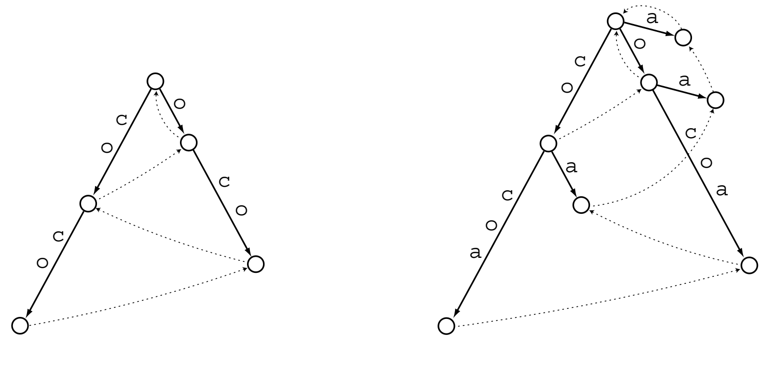


Figure 2.2: $STree(\texttt{coco})$ on the left, and $STree(\texttt{cocoa})$ on the right. The solid arrows represent the edges, while the dotted arrows denote the suffix links.

Theorem 2.1 (McCreight [43]) *Let* $STree(S) = (V, E)$. *Assume* $\|S\| > 1$. *Then* $|V| \leq 2\|S\| + |S|$ *and* $|E| \leq 2\|S\| + |S| - 1$.

2.3.3 DAWGs

Definition 2.9 *$DAWG(S)$ is the directed acyclic graph (V, E) such that*

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

and its suffix links are the set

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

The node $[\varepsilon]^R_S$ is called the *source* node. A node of out-degree zero is called a *sink* node of $DAWG(S)$. If $\mathit{Suffix}(S) - \{\varepsilon\}$ has the prefix property, then there exactly exist $\|S\|$ sink nodes, each of which represents $\mathit{Suffix}(w)$ for each $w \in S$. If $|S| = 1$, $DAWG(S)$ is also written as $DAWG(w)$ where $S = \{w\}$.

We define the *length* of a node $[x]^R_S$ by $|\overset{S}{\overleftarrow{x}}|$. Suppose that $\overset{S}{\overleftarrow{x}} \cdot a \in [y]^R_S$ for $x, y \in \mathit{Factor}(S)$ and $a \in \Sigma$. If $\mathit{length}([y]^R_S) = \mathit{length}([x]^R_S) + |a| = \mathit{length}([x]^R_S) + 1$ (in other words, if $\overset{S}{\overleftarrow{x}} \cdot a = \overset{S}{\overleftarrow{y}}$), the edge $([x]^R_S, a, [y]^R_S)$ is said to be *solid*. Otherwise, it is said to be *non-solid*. For example, in $DAWG(w)$ of Figure 2.3 where $w = \texttt{coco}$, edge $([\texttt{c}]^R_w, \texttt{o}, [\texttt{co}]^R_w)$ is solid, whereas edge $([\varepsilon]^R_w, \texttt{o}, [\texttt{co}]^R_w)$ is non-solid.

As seen in the definition, each node of $DAWG(S)$ is a non-degenerate equivalence class with respect to $\equiv^R_S$. One can see that nodes of $\mathit{STrie}(\texttt{cocoa})$ are 'merged' by the equivalence class under $\equiv^R_S$. In this sense, $DAWG(S)$ can be seen as the minimized version of $\mathit{STrie}(S)$ with "$[(\cdot)]^R_S$ operation".

Suppose that ax is the shortest member of $[ax]^R_S$, for some character $a \in \Sigma$ and string $x \in \mathit{Factor}(S)$. Then the suffix link of node $[ax]^R_S$ in $DAWG(S)$ points to the node $[x]^R_S$ (for example, see nodes $[\texttt{oco}]^R_S$ and $[\texttt{co}]^R_S$ of $DAWG(\texttt{cocoa})$ in Figure 2.3).

Theorem 2.2 (Blumer et al. [8]) *Let $DAWG(S) = (V, E)$. Assume $\|S\| > 1$. Then $|V| \leq 2\|S\| - 1$ and $|E| \leq 3\|S\| - 3$.*

2.3.4 CDAWGs

Definition 2.10 *$CDAWG(S)$ is the directed acyclic graph (V, E) such that*

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

and its suffix links are the set

$$F = \{([\overset{S}{\overrightarrow{ax}}]^R_S, [\overset{S}{\overrightarrow{x}}]^R_S) \mid x, ax \in \mathit{Factor}(S),\ a \in \Sigma,\ \overset{S}{\overrightarrow{ax}} = a \cdot \overset{S}{\overrightarrow{x}},\ \text{and } [\overset{S}{\overrightarrow{x}}]^R_S \neq [\overset{S}{\overrightarrow{ax}}]^R_S\}.$$

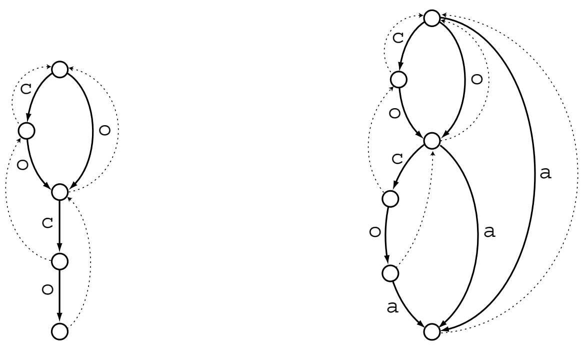


Figure 2.3: *DAWG*(coco) on the left, and *DAWG*(cocoa) on the right. The solid arrows represent the edges, while the dotted arrows denote the suffix links.

The node $[\overset{S}{\overrightarrow{\varepsilon}}]^R_S$ is called the *source* node. A node of out-degree zero is called a *sink* node of *CDAWG*(S). If *Suffix*$(S) - \{\varepsilon\}$ has the prefix property, then there exactly exist $\|S\|$ sink nodes, each of which represents *Suffix*(w) for each $w \in S$. If $|S| = 1$, *CDAWG*(S) is also written as *CDAWG*(w) where $S = \{w\}$.

We define the *length* of a node $[\overset{S}{\overrightarrow{x}}]^R_S$ by $\left|\overset{S}{\overleftarrow{(\overset{S}{\overrightarrow{x}})}}\right| = |\overset{S}{\overleftrightarrow{x}}|$. Suppose that $\overset{S}{\overleftrightarrow{x}} \cdot \alpha \in [\overset{S}{\overrightarrow{y}}]^R_S$ for $x, y \in \mathit{Factor}(S)$ and $\alpha \in \Sigma^*$. If $\mathit{length}([\overset{S}{\overrightarrow{y}}]^R_S) = \mathit{length}([\overset{S}{\overrightarrow{x}}]^R_S) + |\alpha|$ (in other words, if $\overset{S}{\overleftrightarrow{x}} \cdot \alpha = \overset{S}{\overleftrightarrow{y}}$), the edge $([\overset{S}{\overrightarrow{x}}]^R_S, \alpha, [\overset{S}{\overrightarrow{y}}]^R_S)$ is said to be *solid*. Otherwise, it is *non-solid*. In *CDAWG*(w) of Figure 2.4 where $w = \mathtt{coco}$, edge $([\overset{w}{\overrightarrow{\varepsilon}}]^R_w, \mathtt{co}, [\overset{w}{\overrightarrow{\mathtt{co}}}]^R_w)$ is solid, while edge $([\overset{w}{\overrightarrow{\varepsilon}}]^R_w, \mathtt{o}, [\overset{w}{\overrightarrow{\mathtt{co}}}]^R_w)$ is non-solid.

It follows from the definition that *CDAWG*(S) is the minimization of *STree*(S) with "$[(\cdot)]^R_S$ operation". In fact, *CDAWG*(cocoa) in Figure 2.4 can be obtained by 'merging' the isomorphic subtrees in *STree*(cocoa). Similarly, *CDAWG*(S) can also be seen as the compaction of *DAWG*(S) with "$\overset{S}{\overrightarrow{(\cdot)}}$ operation", as seen in *DAWG*(cocoa) in Figure 2.3 and *CDAWG*(cocoa).

Suppose that $ay = \overset{S}{\overrightarrow{ax}}$ is the shortest member of $[\overset{S}{\overrightarrow{ax}}]^R_S$ for some character $a \in \Sigma$ and strings $x, y \in \mathit{Factor}(S)$. Then the suffix link of node $[\overset{S}{\overrightarrow{ax}}]^R_S$ points to the node $[y]^R_S$, where $y = \overset{S}{\overrightarrow{y}}$.

Theorem 2.3 (Blumer et al. [8]) *Let CDAWG*$(S) = (V, E)$. *Assume* $\|S\| > 1$. *Then* $|V| \leq \|S\| + |S|$ *and* $|E| \leq 2\|S\| + |S| - 1$.

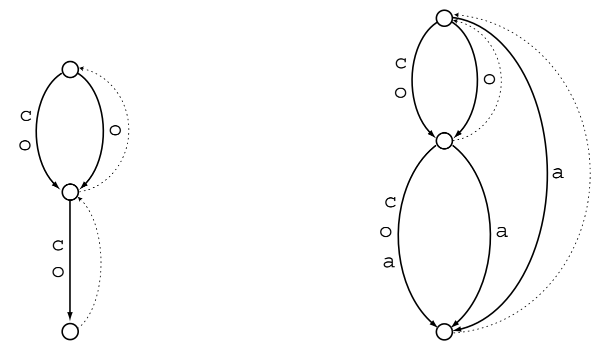


Figure 2.4: *CDAWG*(coco) on the left, and *CDAWG*(cocoa) on the right. The solid arrows represent the edges, while the dotted arrows denote the suffix links.

2.3.5 Suffix Trees Redefined

For any non-empty set S of strings, let $STree'(S)$ denote the tree obtained by removing all internal nodes of out-degree one from $STree(S)$. As seen in Figure 2.5, nodes $\overset{w}{\overrightarrow{\texttt{co}}}$ and $\overset{w}{\overrightarrow{\texttt{o}}}$ in $STree(\texttt{coco})$ are omitted in $STree'(\texttt{coco})$ together with their suffix links. Ukkonen's suffix tree construction algorithm [55] builds $STree'(S)$, not $STree(S)$. The following preparation is necessary for the formal definition of $STree'(S)$.

We introduce a relation X_S over Σ^* such that

$$X_S = \big\{(x, xa) \big| x \in \mathit{Factor}(S) \text{ and } a \in \Sigma \text{ is the unique character such that } xa \in \mathit{Factor}(S)\big\},$$

and let $\equiv'^L_S$ be the equivalence closure of X_S, i.e., the smallest superset of X_S that is symmetric, reflexive, and transitive. It can be readily shown that $\equiv^L_S$ is a refinement of $\equiv'^L_S$, namely, every equivalence class under $\equiv'^L_S$ is a union of one or more equivalence classes in $\equiv^L_S$. For a string $x \in \mathit{Factor}(S)$, let $\overset{S}{\overset{\Longrightarrow}{x}}$ denote the longest string in the equivalence class to which x belongs under the equivalence relation $\equiv'^L_S$.

Proposition 2.1 *For any string $x \in \Sigma^*$, $\overset{S}{\overrightarrow{x}}$ is a prefix of $\overset{S}{\overset{\Longrightarrow}{x}}$. If $\overset{S}{\overrightarrow{x}} \neq \overset{S}{\overset{\Longrightarrow}{x}}$, then $\overset{S}{\overrightarrow{x}} \in$ Suffix(S).*

Proposition 2.2 *If set Suffix$(S) - \{\varepsilon\}$ satisfies the prefix property, $\overset{S}{\overset{\Longrightarrow}{x}} = \overset{S}{\overrightarrow{x}}$ for any string $x \in$ Factor(S).*

We are now ready to define $STree'(S)$.

Definition 2.11 *STree′(S) is the tree* (V, E) *such that*

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

and its suffix links are the set

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

This definition is the same as the one obtained by replacing "$\overset{S}{\overrightarrow{(\cdot)}}$ operation" with "$\overset{S}{\overset{\Longrightarrow}{(\cdot)}}$ operation" in Definition 2.8.

Corollary 2.2 *If Suffix(S)* $- \{\varepsilon\}$ *has the prefix property, STree′(S) = STree(S).*

As previously stated, Ukkonen's algorithm constructs *STree′(S)*. Even in case the set *Suffix(S)* $- \{\varepsilon\}$ does not have the prefix property, *STree′(S′)* = *STree(S′)* where $S' = \{w_i\$_i \mid w_i \in S$ and $\$_i \notin \mathit{Factor}(S)$ for $1 \leq i \leq |S|\}$.

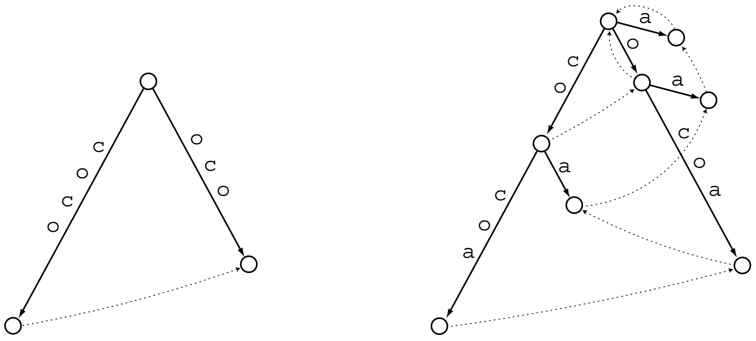


Figure 2.5: *STree′*(`coco`) on the left, and *STree′*(`cocoa`) on the right. The solid arrows represent the edges, while the dotted arrows denote the suffix links.

2.3.6 CDAWGs Redefined

Similarly to *STree′(S)*, for any non-empty set S of strings, *CDAWG′(S)* has no internal node of out-degree one. Our algorithm to be introduced in Section 3.4 and Section 3.5 constructs *CDAWG′(S)*.

Definition 2.12 *$CDAWG'(S)$ is the directed acyclic graph (V, E) such that*

$$\begin{aligned} V &= \{[\overset{S}{\overset{\Longrightarrow}{x}}]^R_S \mid x \in \mathit{Factor}(S)\}, \\ E &= \{([\overset{S}{\overset{\Longrightarrow}{x}}]^R_S, a\beta, [\overset{S}{\overset{\Longrightarrow}{xa}}]^R_S) \mid x, xa \in \mathit{Factor}(S),\ a \in \Sigma,\ \beta \in \Sigma^*,\ \overset{S}{\overset{\Longrightarrow}{xa}} = xa\beta, \text{ and } \overset{S}{\overset{\Longrightarrow}{x}} \neq \overset{S}{\overset{\Longrightarrow}{xa}}\}, \end{aligned}$$

and its suffix links are the set

$$F = \{([\overset{S}{\overset{\Longrightarrow}{ax}}]^R_S, [\overset{S}{\overset{\Longrightarrow}{x}}]^R_S) \mid x, ax \in \mathit{Factor}(S),\ a \in \Sigma,\ \overset{S}{\overset{\Longrightarrow}{ax}} = a \cdot \overset{S}{\overset{\Longrightarrow}{x}},\ \text{and } [\overset{S}{\overset{\Longrightarrow}{x}}]^R_S \neq [\overset{S}{\overset{\Longrightarrow}{ax}}]^R_S\}.$$

As in case of $STree'(S)$ and $STree(S)$, the definition equals the one obtained by substituting "$\overset{S}{\overrightarrow{(\cdot)}}$ operation" with "$\overset{S}{\overset{\Longrightarrow}{(\cdot)}}$ operation" in Definition 2.10.

Corollary 2.3 *If $\mathit{Suffix}(S) - \{\varepsilon\}$ has the prefix property, $CDAWG'(S) = CDAWG(S)$.*

Even in case that $\mathit{Suffix}(S) - \{\varepsilon\}$ does not have the prefix property, $CDAWG'(S') = CDAWG(S')$ where $S' = \{w_i\$_i \mid w_i \in S \text{ and } \$_i \notin \mathit{Factor}(S) \text{ for } 1 \leq i \leq |S|\}$.

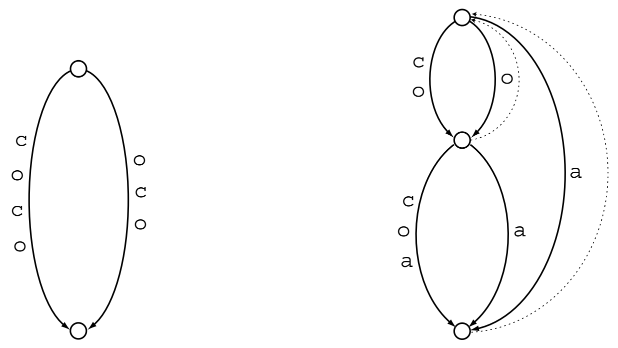


Figure 2.6: $CDAWG'$(coco) on the left, and $CDAWG'$(cocoa) on the right. The solid arrows represent the edges, while the dotted arrows denote the suffix links.

Chapter 3

On-Line Construction of Compact Directed Acyclic Word Graphs

3.1 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. The most widely known and studied structure of this kind seems to be the suffix tree [58, 43, 11, 55, 20, 38, 54], perhaps because there are a "myriad" [2] of applications for it. For any string w the suffix tree of w requires only $O(n)$ space and can be built in $O(n)$ time, where n is the length of w. Although its theoretical space complexity is linear, much attention has been devoted to the reduction of the practical space requirement of the structure. This has led to the introduction of more space-economical index structures, like suffix arrays [41], suffix cacti [35], compact suffix arrays [40], compressed suffix arrays [19, 47], and so on.

Blumer et al. [7] introduced the directed acyclic word graph (DAWG) for a string, which is the smallest finite state automaton to recognize all suffixes of the string [12]. DAWGs are also involved in several combinatorial algorithms on strings [14, 8, 24, 5, 56], since they serve as indexes of the string, as well as other index structures such as suffix tries and suffix trees.

In this work, we focus our attention on the compact directed acyclic word graph (CDAWG) first introduced by Blumer et al. in [8]. Crochemore and Vérin displayed a relationship among suffix tries, suffix trees, DAWGs, and CDAWGs [16]. Suffix trees (resp. DAWGs) are the compacted (resp. minimized) version of suffix tries, as shown

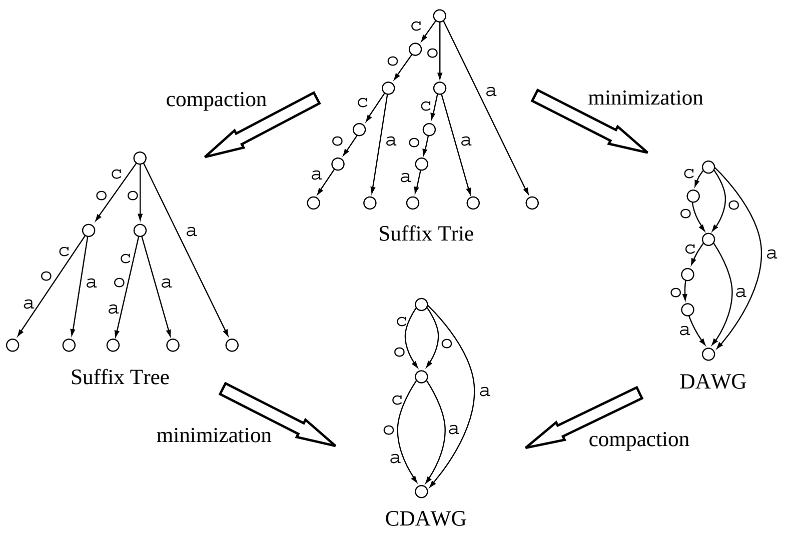


Figure 3.1: Relationship among the suffix trie, the suffix tree, the DAWG, and the CDAWG for string cocoa.

in Figure 3.1. Similarly, CDAWGs can be obtained by either compacting DAWGs or minimizing suffix trees.

Not only in theory as stated above, but also in practice, CDAWGs provide significant reductions of the memory space required by suffix trees and DAWGs, as experimental results have shown in [8, 16]. In Bioinformatics a considerable amount of DNA sequences has to be processed efficiently, both in space and 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 in [7]. 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 [16]: a linear-time algorithm to construct the CDAWG for a string *directly*. Their algorithm is based on McCreight's suffix tree construction algorithm [43]. 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, if a new character is added to the input string. Table 3.1 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 chapter, we present a new linear-time algorithm to directly construct the

Index Structure	*Algorithm*	*linear time*	*on-line*	*multiple strings*
suffix tries	Ukkonen [55]		√	√
suffix trees	Weiner [58]	√		
	McCreight [43]	√		
	Ukkonen [55]	√	√	√
DAWGs	Blumer et al. [7]	√	√	
	Blumer et al. [8]	√	√	√
CDAWGs	Blumer et al. [8]	√		√
	Crochemore and Vérin [16]	√		

Table 3.1: The properties of algorithms for the construction of index structures.

CDAWG for a given string, which is based on Ukkonen's suffix tree construction algorithm [55]. 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 [16], 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*. The CDAWG for a set of strings can be constructed by the algorithm given in [8] 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.

3.2 On-Line Construction of the Suffix Trie for a Single String

The on-line CDAWG construction algorithm we will give later on is based on Ukkonen's on-line suffix tree construction algorithm [55]. Moreover, Ukkonen's algorithm is based on an intuitive on-line algorithm that constructs suffix tries. We firstly consider the case that we are given a single string as an input for the algorithm.

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 2.7 that $\mathit{suf}(x) = y$ for some $y \in \mathit{Factor}(w)$ such that $x = ay$ for some character $a \in \Sigma$. For the case that $x = \varepsilon$, let $\mathit{suf}(\varepsilon) = \perp$ where $\perp$ is an auxiliary node called the *bottom* node. We suppose that there exists an edge $(\perp, \Sigma, \varepsilon)$, where the symbol Σ here means every character in the alphabet. Assuming that the bottom node $\perp$ corresponds to the inverse a^{-1} for any $a \in \Sigma$, the edge $(\perp, \Sigma, \varepsilon)$ 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.

The algorithm reads a given string $w \in \Sigma^*$ from left to right, while building $\mathit{STrie}(w[1:i])$ for $1 \leq i \leq |w|$. It is easy to construct $\mathit{STrie}(w[1:i+1])$ by updating $\mathit{STrie}(w[1:i])$. What is necessary here is to insert suffixes of $w[1:i+1]$ into $\mathit{STrie}(w[1:i])$.

Definition 3.1 *For an arbitrary string $u \in \Sigma^*$ and an arbitrary character $a \in \Sigma$, the* longest repeated suffix *(the* LRS *for short) of ua is the longest element of the set $\mathit{Factor}(u) \cap \mathit{Suffix}(ua)$.*

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

The suffixes of $w[1:i+1]$ can be divided into the following two groups, by the LRS of $w[1:i+1]$.

(1) Suffixes $w[h:i+1]$ for $1 \leq h \leq j$ where $w[j+1:i+1]$ is the LRS of $w[1:i+1]$.

(2) Suffixes $w[h':i+1]$ for $j+1 \leq h' \leq i+2$.

The group (2) is empty in case the LRS of $w[1:i+1] = \varepsilon$, that is, in case $j+1 = i+2$.

There is no need to newly insert any suffixes in the group (2), simply because they have already been represented in $\mathit{STree}'(w[1:i])$. The algorithm creates a new node corresponding to $w[h:i+1]$ for each h $(1 \leq h \leq j)$, together with a new edge $(w[h:i], w[i+1], w[h:i+1])$, by traversing $\mathit{suf}(w[h:i])$ to move to the next node $w[h-1:j]$. When it finds the node corresponding to the LRS $w[j+1:i]$, the algorithm stops and the update then gets completed. The node with respect to the LRS $w[j+1:i+1]$ is called the *end point* of $\mathit{STrie}(w[1:i+1])$.

The on-line construction of $\mathit{STrie}(\texttt{cocoa})$ is shown in Figure 3.2.

Theorem 3.1 (Ukkonen [55]) *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.*

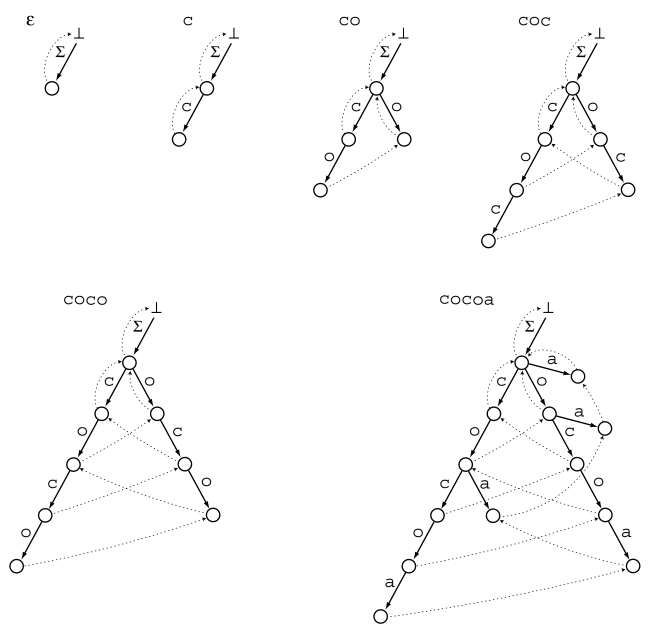


Figure 3.2: On-line construction of *STrie*(w) with $w = \mathsf{cocoa}$.

3.3 On-Line Construction of the Suffix Tree for a Single String

3.3.1 Informal Description

We firstly summarize Ukkonen's suffix tree construction algorithm in the comparison with the previous suffix trie algorithm. Figure 3.3 shows the on-line construction of *STree*′(cocoa). Focus on the update of *STree*′(co) to *STree*′(coc). Differently from that of *STrie*(co) to *STrie*(coc), the edges leading to leaf nodes are *automatically* extended with the new character c in *STree*′(coc). This is feasible by the idea so-called *open edges*.

See the first and second steps of the update of *STree*′(coco) to *STree*′(cocoa). The gray star mark indicates the *active point* from which a new edge is created in each step. After the new edge (co, a, 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 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 suffix links of leaf nodes are never utilized in Ukkonen's algorithm, it does not create any of them.

3.3.2 Ukkonen's Algorithm

Ukkonen's on-line suffix tree construction algorithm is based on the on-line algorithm to build suffix tries recalled in Section 3.2. As stated in Definition 2.11, an edge of $STree'(w)$ is labeled by a string $\alpha \in Factor(w)$. The key to achieve a linear-space implementation of the suffix tree is to label the edge $(\overset{w}{\Longrightarrow}{x}, \alpha, \overset{w}{\Longrightarrow}{x\alpha})$ in $STree'(w)$ by (k, p), such that $w[k : p] = \alpha$.

An implicit node $y \in Factor(w)$ can be represented by a pair $(\overrightarrow{x}^{w}, \alpha)$ of an explicit node $\overrightarrow{x}^{w}$ and a string $\alpha \in Factor(w)$ such that $y = \overrightarrow{x}^{w} \cdot \alpha$. The pair $(\overrightarrow{x}^{w}, \alpha)$ 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 $(\overrightarrow{x}^{w}, \alpha)$ for y in which $|\alpha|$ is minimized is called the *canonical* reference pair for y. The reference pair can also be written as $(\overrightarrow{x}^{w}, (k, p))$ such that $w[k : p] = \alpha$.

Ukkonen's algorithm reads a given string $w \in \Sigma^*$ from left to right, while building $STree'(w[1 : i])$ for $1 \le i \le |w|$. Suppose that we from now on update $STree'(w[1 : i])$ to $STree'(w[1 : i+1])$. The group (1) of the suffixes of $w[1 : i+1]$, mentioned in the previous section, can moreover be divided into two as follows by integer j'.

(1-a) Suffixes $w[l : i + 1]$ for $1 \le l \le j'$ where $w[j' + 1 : i]$ is the LRS of $w[1 : i]$.

(1-b) Suffixes $w[\ell : i + 1]$ for $j' + 1 \le \ell \le j$.

We remark that all the suffixes of the group (1-a) are those represented by leaf nodes in $STree'(w[1 : i])$. Note that, for any l, $\overset{w[1:i]}{\overrightarrow{w[l : i]}} = w[l : i]$ and $\overset{w[1:i+1]}{\overrightarrow{w[l : i + 1]}} = w[l : i + 1]$. That is, intuitively, every leaf node of $STree'(w[1 : i])$ is also a leaf node in $STree'(w[1 : i + 1])$. This fact is crucial to Ukkonen's algorithm in order that it *automatically* inserts those in the group (1-a) into $STree'(w[1 : i + 1])$, by means of *open edges*.

Suppose that $(\overset{w[1:i]}{\overrightarrow{x}}, \alpha, \overset{w[1:i]}{\overrightarrow{x\alpha}})$ is an edge of $STree'(w[1 : i])$ where $\overset{w[1:i]}{\overrightarrow{x\alpha}}$ is a leaf node. Letting k be the integer such that $w[k : i] = \alpha$, it is feasible to label the edge by (k, ∞). This way we need no explicit insertion of the suffixes of $w[1 : i + 1]$ in the group (1-a).

The location from which a suffix $w[\ell : i+1]$ with respect to the group (1-b) is inserted is called the *active point* of $STree'(w[1 : i+1])$. The active point for $w[1 : i+1]$ begins at the node $w[j'+1:i]$, where $w[j'+1:i]$ is the end point of $STree'(w[1:i])$. Assume we are now inserting suffix $w[\ell:i+1]$ into $STree'(w[1:i])$, where $j'+1 \leq \ell \leq j$. There are two cases to consider for the active point.

(Case 1) The active point is on an explicit node $w[\ell:i]$. In this case,

$$\overset{w[1:i]}{\overrightarrow{w[\ell:i]}} = \overset{w[1:i+1]}{\overrightarrow{w[\ell:i]}} = w[\ell:i].$$

Let $x = w[\ell:i]$. In this case a new edge $(\overset{w[1:i+1]}{\overrightarrow{x}}, \alpha, \overset{w[1:i+1]}{\overrightarrow{x\alpha}})$ is created, where $\alpha = w[i+1:i+1]$. Note $\overset{w[1:i+1]}{\overrightarrow{x\alpha}} = w[\ell:i+1]$. The edge is actually labeled by $(i+1,\infty)$. After that, the active point moves to the explicit node $\mathit{suf}(\overset{w[1:i+1]}{\overrightarrow{x}})$, corresponding to $w[\ell-1:i]$, in order to insert the next suffix $w[\ell-1:i+1]$.

(Case 2) The active point is on an implicit node $w[\ell:i]$. In this case,

$$\overset{w[1:i]}{\overrightarrow{w[\ell:i]}} \neq w[\ell:i] \text{ but } \overset{w[\ell:i+1]}{\overrightarrow{w[\ell:i]}} = w[\ell:i].$$

Let $(\overset{w[1:i]}{\overrightarrow{x}}, \alpha)$ be the canonical reference pair for the active point, namely, $\overset{w[1:i]}{\overrightarrow{x}} \cdot \alpha = w[\ell:i]$. Focus on the edge $(\overset{w[1:i]}{\overrightarrow{x}}, \alpha\beta, \overset{w[1:i]}{\overrightarrow{x\alpha\beta}})$ where $\beta \neq \varepsilon$. The edge is replaced by the edges $(\overset{w[1:i+1]}{\overrightarrow{x}}, \alpha, \overset{w[1:i+1]}{\overrightarrow{x\alpha}})$ and $(\overset{w[1:i+1]}{\overrightarrow{x\alpha}}, \beta, \overset{w[1:i+1]}{\overrightarrow{x\alpha\beta}})$ where $\overset{w[1:i+1]}{\overrightarrow{x\alpha}}$ is a new explicit node. Then a new edge $(\overset{w[1:i+1]}{\overrightarrow{x\alpha}}, \gamma, \overset{w[1:i+1]}{\overrightarrow{x\alpha\gamma}})$ is created, where $\gamma = w[i+1:i+1]$. Note $\overset{w[1:i+1]}{\overrightarrow{x\alpha\gamma}} = w[\ell:i+1]$. The edge is actually labeled by $(i+1,\infty)$.

After that, we need to move to the (implicit or explicit) node corresponding to $w[\ell-1:i]$, the next active point, but the table *suf* is yet to be computed for the new node $\overset{w[1:i+1]}{\overrightarrow{x\alpha}}$. Thus we once move to its parent node $\overset{w[1:i+1]}{\overrightarrow{x}}$ for which $\mathit{suf}(\overset{w[1:i+1]}{\overrightarrow{x}})$ must have already been computed. Let $\mathit{suf}(\overset{w[1:i+1]}{\overrightarrow{x}}) = \overset{w[1:i+1]}{\overrightarrow{y}}$, where there exists some character a such that $\overset{w[1:i+1]}{\overrightarrow{x}} = a \cdot \overset{w[i+1]}{\overrightarrow{y}}$. Note that $\overset{w[1:i+1]}{\overrightarrow{y}} \cdot \alpha = w[\ell-1:i]$. We go down from the node $\overset{w[1:i+1]}{\overrightarrow{y}}$ with spelling out α, to obtain the canonical reference pair for the active point $w[\ell-1:i]$. The node $w[\ell-1:i]$ either is already, or will in this step become, explicit. The value of $\mathit{suf}(\overset{w[1:i+1]}{\overrightarrow{x\alpha}})$ is then set to $w[\ell-1:i]$. This way the algorithm 'simulates' the suffix-link-traversal of suffix tries.

Figure 3.3 shows the on-line construction of $STree'(\texttt{cocoa})$.

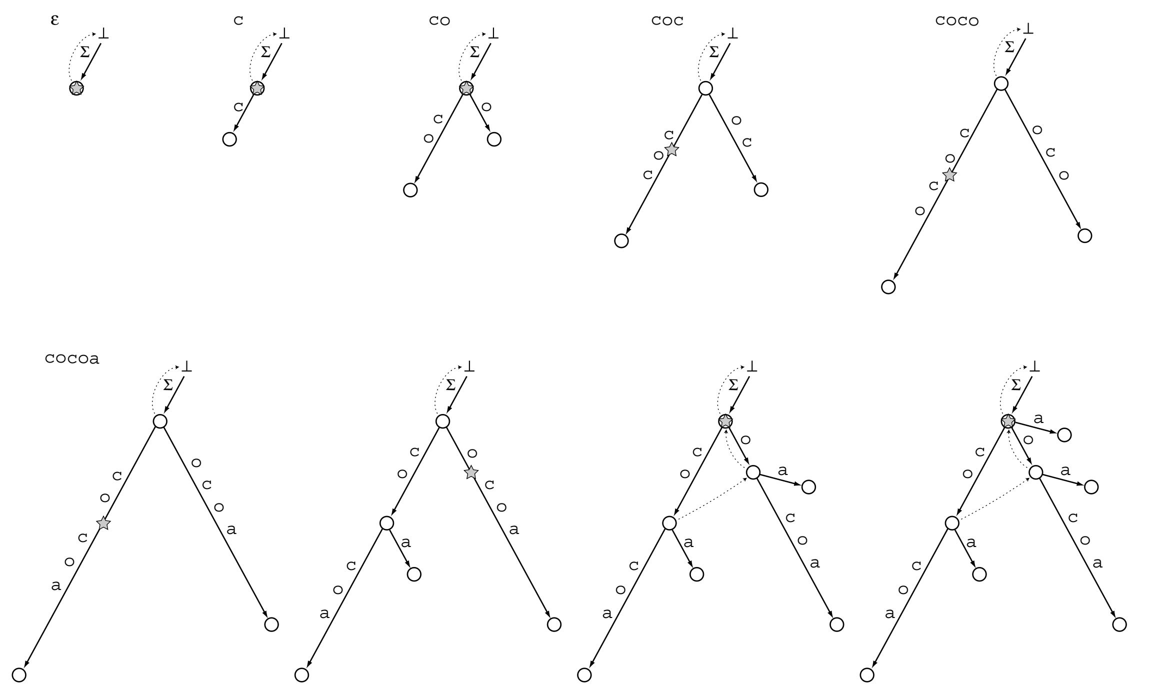


Figure 3.3: On-line construction of *STree*$'(w)$ with $w =$ `cocoa`. The star represents the active point for each step.

A pseudo-code for Ukkonen's algorithm is shown in Fig 3.4. There, 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.

Theorem 3.2 (Ukkonen [55]) *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.*

3.4 On-Line Construction of the CDAWG for a Single String

3.4.1 Informal Description

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 Figure 3.5 that shows the on-line construction of *CDAWG*$'$(`cocoa`), in comparison with Figure 3.3 displaying the on-line construction of *STree*$'$(`cocoa`). Compare *CDAWG*$'$(`co`) and *STree*$'$(`co`). While

```
**Algorithm** for on-line construction of $STree'(w\$)$
in alphabet $\Sigma = \{w[-1], w[-2], \dots w[-m]\}$.
/* $ is the end-marker appearing nowhere in w. */
 1 create nodes root and $\perp$;
 2 for $j := 1$ to $m$ do create edge $(\perp, (-j, -j), root)$;
 3 $suf(root) := \perp$;
 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 $k \le p - 1$ then $r := split\_edge(s, (k, p-1))$;    /* implicit case. */
 4     else $r := s$;    /* explicit case. */
 5     create node $r'$;    create edge $(r, (p, \infty), r')$;
 6     if $oldr \neq$ nil then $suf(oldr) := r$;
 7     $oldr := r$;
 8     $(s, k) := canonize(suf(s), (k, p-1))$;
 9 if $oldr \neq$ nil then $suf(oldr) := s$;
10 return $canonize(s, (k, p))$;

function check_end_point(s, (k, p), c): boolean;
 1 if $k \le p$ 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 $p' - k' \le p - k$ do
 4     $k := k + p' - k' + 1$;    $s := s'$;
 5     if $k \le p$ 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 edges $(s, (k', k' + p - k), r)$ and $(r, (k' + p - k + 1, p'), s')$;
 4 return $r$;
```

Figure 3.4: Ukkonen's on-line algorithm for constructing suffix trees.

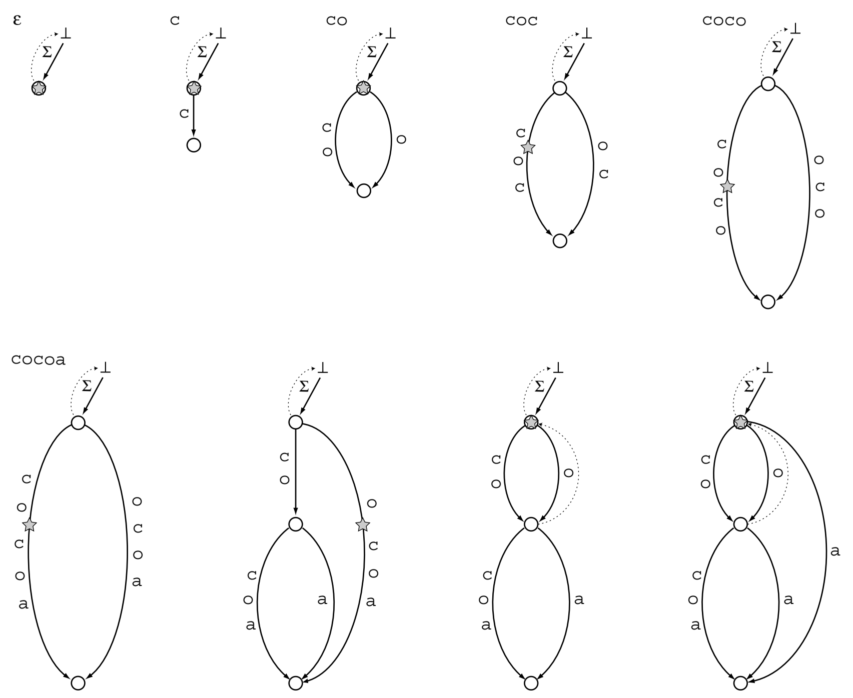


Figure 3.5: On-line construction of *CDAWG*$'(w)$ with $w = \texttt{cocoa}$. The star mark represents the active point for each step.

strings `co` and `o` are separately represented in *STree*$'(\texttt{co})$, they are in the same node in *CDAWG*$'(\texttt{co})$. The destination 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 at the second step 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. 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 equivalent relation $\equiv^R_{\texttt{cocoa}}$, they are *merged* into a single node in *CDAWG*$'(\texttt{cocoa})$.

3.4.2 The Algorithm

The algorithm presented in this section for the on-line construction of CDAWGs behaves similarly to Ukkonen's algorithm. Let $u = w[1:i]$ and $ua = w[1:i+1]$, namely, $a = w[i+1]$. The difference between them is summarized as follows.

- All the suffixes in the group (1) are equivalent under $\equiv^R_{ua}$. Thus all of them are represented in the sink node $[\overset{ua}{\overrightarrow{ua}}]^R_{ua}$. Namely, the destinations of the open edges are all the same. According to this property, we can generalize the idea of open edges as follows. For any open edge $(s, (k, \infty), t)$ of $CDAWG'(w)$ where t denotes the sink node $[\overset{ua}{\overrightarrow{ua}}]^R_{ua}$, we actually implement it as $(s, (k, e), t)$ where e is a global variable that denotes $|ua|$. Thus, when a new character added after u, we can extend all open edges only with increasing the value of e by 1. Obviously, it only takes $O(1)$ time.
- Consider **(Case 2)**. There can be integers ℓ_1, ℓ_2 with $j' + 1 \leq \ell_1 < \ell_2 \leq j$ such that $w[\ell_1 : i] \equiv^R_{ua} w[\ell_2 : i]$. In such case, they are *merged* into a single explicit node $[\overset{ua}{\overrightarrow{w[\ell_1 : i]}}]^R_{ua}$, during the update of $CDAWG'(u)$ to $CDAWG'(ua)$. The equivalence test is performed on the basis of Lemma 3.1 to be given in the sequel.
- Consider strings $x, y \in Factor(u)$ such that $\overset{u}{\overrightarrow{x}} = x$ and $\overset{u}{\overrightarrow{y}} = y$. Assume that $x \equiv^R_u y$, that is, they are represented in the same explicit node $[x]^R_u$ in $CDAWG'(u)$. Note that, however, x, y might not be equivalent under $\equiv^R_{ua}$. When $CDAWG'(u)$ is updated to $CDAWG'(ua)$, then the node has to be *separated* into two nodes $[x]^R_{ua}$ and $[y]^R_{ua}$. Since this node separation happens only when $x \notin Suffix(ua)$ but $y \in Suffix(ua)$, we can do this procedure after we find the end point. The condition of the node separation will be given later on, in Lemma 3.2.

Merging Implicit Nodes.

As mentioned above, it can happen that two or more nodes implicit in $CDAWG'(u)$ are merged into one explicit node in $CDAWG'(ua)$. As a concrete example, we show in Figure 3.7 the snapshot of the conversion of $CDAWG'(u)$ into $CDAWG'(ua)$ with $u =$ `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. The examination whether to merge implicit nodes can be done by testing the equivalence of two nodes under the equivalence relation $\equiv^R_{ua}$. The equivalence test can be performed on the basis of the following proposition and lemma.

Proposition 3.1 *Let $x \in Factor(w)$ for a string w, and let $z = \overset{w}{\overleftrightarrow{x}}$. Then, string x occurs within string z exactly once.*

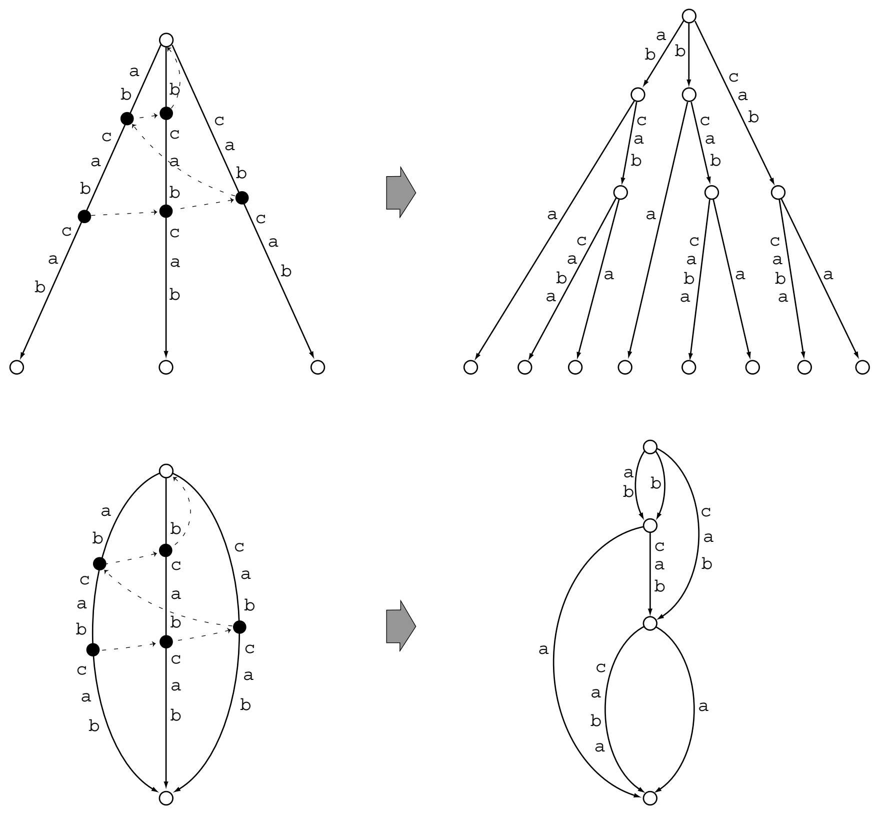


Figure 3.6: Comparison of conversions. One is from *STree*′(u) to *STree*′(ua), while the other is from *CDAWG*′(u) to *CDAWG*′(ua) for $u =$ `abcabcab` and $a =$ `a`. The black circles represent implicit nodes to be merged in the next step, connected by implicit suffix links corresponding to the traversal by the active point.

Lemma 3.1 *Let $w \in \Sigma^*$. For any strings $x, y \in$ Factor(w) with $y \in$ Suffix(x),*

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

Proof. If $x \equiv^R_w y$, we have $\overset{w}{\overleftarrow{x}} = \overset{w}{\overleftarrow{y}}$ by Definition 2.3. By Corollary 2.1, we know $\overset{w}{\overrightarrow{(\overset{w}{\overleftarrow{x}})}} = \overset{w}{\overleftarrow{(\overset{w}{\overrightarrow{x}})}}$ and $\overset{w}{\overrightarrow{(\overset{w}{\overleftarrow{y}})}} = \overset{w}{\overleftarrow{(\overset{w}{\overrightarrow{y}})}}$, which yield $\overset{w}{\overleftarrow{(\overset{w}{\overrightarrow{x}})}} = \overset{w}{\overleftarrow{(\overset{w}{\overrightarrow{y}})}}$. Again by Definition 2.3, we have $[\overset{w}{\overrightarrow{x}}]^R_w = [\overset{w}{\overrightarrow{y}}]^R_w$.

Conversely, suppose $[\overset{w}{\overrightarrow{x}}]^R_w = [\overset{w}{\overrightarrow{y}}]^R_w$. Recall that $\overset{w}{\overleftrightarrow{x}} = \overset{w}{\overleftarrow{(\overset{w}{\overrightarrow{x}})}}$ by Corollary 2.1 and $\overset{w}{\overleftarrow{(\overset{w}{\overrightarrow{x}})}}$ is the unique longest member of $[\overset{w}{\overrightarrow{x}}]^R_w$. Similarly, $\overset{w}{\overleftrightarrow{y}}$ is the unique longest member of $[\overset{w}{\overrightarrow{y}}]^R_w$. Thus we have $\overset{w}{\overleftrightarrow{x}} = \overset{w}{\overleftrightarrow{y}}$. Let $z = \overset{w}{\overleftrightarrow{x}} = \overset{w}{\overleftrightarrow{y}}$. 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 3.1. Since $\overset{w}{\overleftrightarrow{y}} = \alpha\delta y\beta$, we conclude that every occurrence of y within w must be preceded by δ. Thus we have

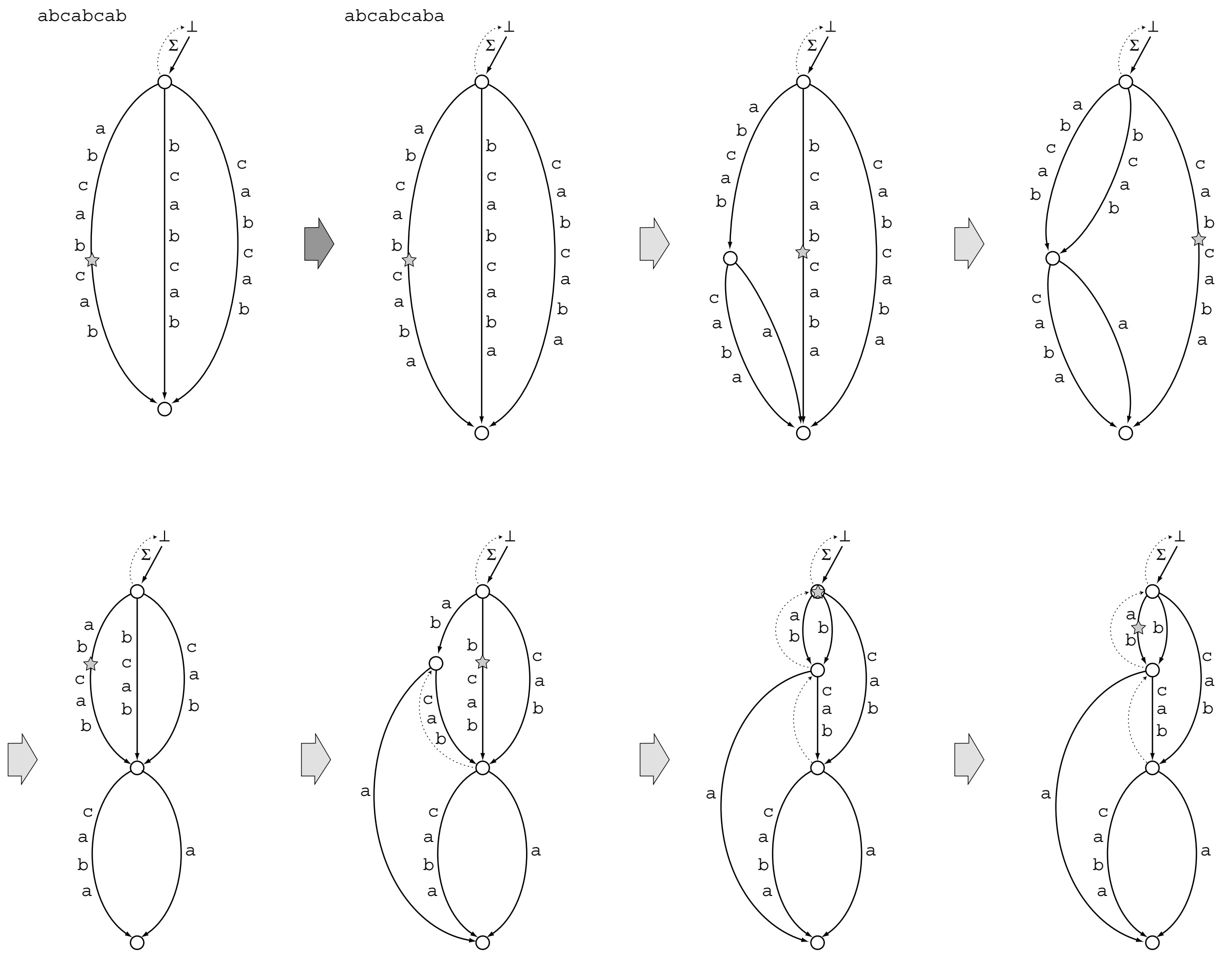


Figure 3.7: Detailed conversion from $CDAWG'(u)$ to $CDAWG'(ua)$ for $u = \texttt{abcabcab}$ and $a = \texttt{a}$.

$x \equiv^R_w y$. $\square$

For any string $x \in \mathit{Factor}(w)$, the equivalence class $[\overset{w}{\overrightarrow{x}}]^R_w$ is the closest explicit child of the node for x in $\mathit{CDAWG}(w)$. Thus we can test the equivalence of two suffixes x, y of w with Lemma 3.1.

The matter is that, for a string $v \in \mathit{Suffix}(w)$, the node $\overset{w}{\overrightarrow{v}}$ might not be explicit in $\mathit{CDAWG}'(w)$. Namely, on the equivalence test, we might refer to the node $[\overset{w}{\Longrightarrow}\!\!\!x]^R_w$ instead of $[\overset{w}{\overrightarrow{x}}]^R_w$. Nevertheless, it does not actually happen in our on-line manner in which suffixes are processed in decreasing order of their length.

See $\mathit{CDAWG}'(u)$ shown on the right of Figure 3.6, where $u = \texttt{abcabcab}$. The black points are the implicit nodes the active point traverses in the next step via 'implicit' suffix links. In $\mathit{CDAWG}'(u)$, $[\overset{u}{\Longrightarrow}\!\!\!\!\!\texttt{cab}]^R_u = [\overset{u}{\Longrightarrow}\!\!\!\!\texttt{ab}]^R_u = [\overset{u}{\Longrightarrow}\!\!\!u]^R_u$. However, in $\mathit{CDAWG}'(ua)$, $\texttt{cab} \not\equiv^R_{ua} \texttt{ab}$ where $a = \texttt{a}$. See Figure 3.7 in which the detail of the update of $\mathit{CDAWG}'(u)$ to $\mathit{CDAWG}'(ua)$ is displayed. Notice that there is no trouble on merging the implicit nodes.

Separating Explicit Nodes.

When $\mathit{CDAWG}'(u)$ is updated to $\mathit{CDAWG}'(ua)$, an explicit node $[\overset{u}{\Longrightarrow}\!\!\!x]^R_u$ with $x \in \mathit{Factor}(u)$ might be separated into two explicit nodes $[\overset{ua}{\Longrightarrow}\!\!\!x]^R_{ua}$ and $[\overset{ua}{\Longrightarrow}\!\!\!y]^R_{ua}$ if $x \notin \mathit{Suffix}(ua)$, $y \in \mathit{Suffix}(x)$, and $y \in \mathit{Suffix}(ua)$. It is inherently the same 'phenomenon' as the node separation occurring in the on-line construction of DAWGs [7]. Therefor we briefly recall the essence of the node separation of DAWGs. For $u \in \Sigma^*$ and $a \in \Sigma$, $\equiv^R_{ua}$ is a refinement of $\equiv^R_u$. Furthermore, we have the following lemma.

Lemma 3.2 (Blumer et al. [7]) *Let $u \in \Sigma^*$ and $a \in \Sigma$. Let z be the LRS of ua. For a string $x \in \mathit{Factor}(u)$, assume $x = \overset{u}{\overleftarrow{x}}$. Then,*

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

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

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

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

Figure 3.8 shows the conversion from $DAWG(u)$ to $DAWG(ua)$ with $u = \texttt{cocoa}$ and $a = \texttt{a}$. The LRS of the string $\texttt{cocoao}$ is $\texttt{o}$, therefore we focus on edge $([\varepsilon]^R_u, \texttt{o}, [\texttt{o}]^R_u)$. Since $\mathit{length}([\varepsilon]^R_u) + |\texttt{o}| = 1 < \mathit{length}([\texttt{o}]^R_u) = 2$, node $[\texttt{o}]^R_u$ is separated into two nodes $[\texttt{co}]^R_{ua}$ and $[\texttt{o}]^R_{ua}$, as shown in Figure 3.8.

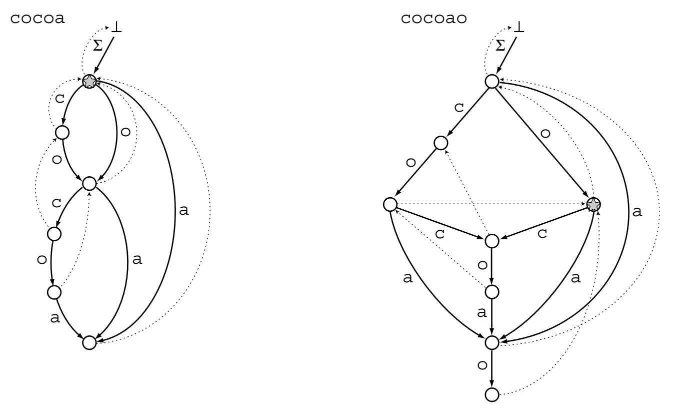


Figure 3.8: The update of $DAWG(u)$ to $DAWG(ua)$, where $u = \texttt{cocoa}$ and $a = \texttt{o}$.

Now we go back to the update of $CDAWG'(u)$ to $CDAWG'(ua)$. The test whether to separate a node when a CDAWG is updated can also be done on the basis of Lemma 3.2 in the very similar way. Since only explicit nodes can be separated, we only need to care about the case that $z = \overset{ua}{\overrightarrow{z}}$ where z is the LRS of ua. It is not difficult to establish the following lemma.

Lemma 3.3 *Let $w \in \Sigma^*$. Assume the LRS of w is z. Then, if $z = \overset{w}{\Rightarrow}{z}$, $\overset{w}{\overrightarrow{x}} = \overset{w}{\overleftrightarrow{x}}$ for any string $x \in \mathit{Factor}(w)$.*

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

$$\begin{aligned} z = \overset{u}{\overleftrightarrow{x}} &\Leftrightarrow \mathit{length}([y]^R_u) + |\alpha| = \mathit{length}([x]^R_u), \text{ and} \\ z \neq \overset{u}{\overleftrightarrow{x}} &\Leftrightarrow \mathit{length}([y]^R_u) + |\alpha| < \mathit{length}([x]^R_u). \end{aligned}$$

Figure 3.9 shows the update of $\mathit{CDAWG}'(u)$ to $\mathit{CDAWG}'(ua)$, where $u = \mathtt{cocoa}$ and $a = \mathtt{o}$. The LRS of the string cocoao is o, therefore we focus on edge $([\overset{u}{\overrightarrow{\varepsilon}}]^R_u, \mathtt{o}, [\overset{u}{\overrightarrow{\mathtt{o}}}]^R_u)$. Since $\mathit{length}([\overset{u}{\overrightarrow{\varepsilon}}]^R_u) + |\mathtt{o}| = 1 < \mathit{length}([\overset{u}{\overrightarrow{\mathtt{o}}}]^R_u) = 2$, node $[\overset{u}{\overrightarrow{\mathtt{o}}}]^R_u$ is separated into two nodes $[\overset{ua}{\overrightarrow{\mathtt{co}}}]^R_{ua}$ and $[\overset{ua}{\overrightarrow{\mathtt{o}}}]^R_{ua}$, as shown in Figure 3.9.

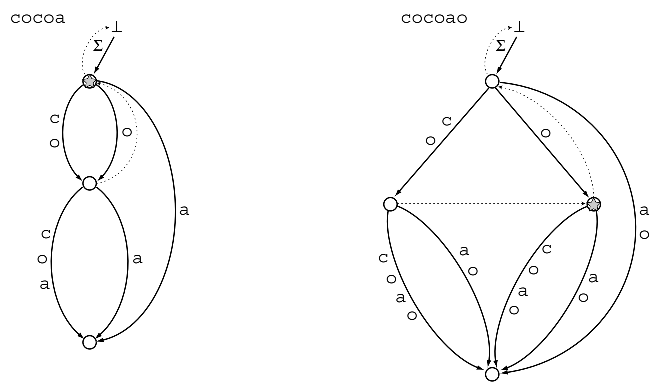


Figure 3.9: The update of $\mathit{CDAWG}'(u)$ to $\mathit{CDAWG}'(ua)$, where $u = \mathtt{cocoa}$ and $a = \mathtt{o}$.

Pseudo-Code.

The algorithm is described in Figure 3.10 and Figure 3.11. 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 the same as the one used in Ukkonen's algorithm, 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 $\mathit{DAWG}(w)$ given by Blumer et al. [7], except that implicit nodes are also treated.

```
Algorithm for on-line construction of $CDAWG'(w\$)$
in alphabet $\Sigma = \{w[-1], w[-2], \ldots, w[-m]\}$.
/* $\$$ is the end-marker appearing nowhere in $w$. */
 1 create nodes source, sink, and $\perp$;
 2 for $j := 1$ to $m$ do create a new edge $(\perp, (-j, -j), source)$;
 3 $suf(source) := \perp$;
 4 $length(source) := 0$;   $length(\perp) := -1$;
 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 $k \leq p - 1$ 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 $oldr \neq$ nil then $suf(oldr) := r$;
15     $oldr := r$;
16     $(s, k) := canonize(suf(s), (k, p-1))$;
17 if $oldr \neq$ nil then $suf(oldr) := s$;
18 return $separate\_node(s, (k, p))$;
```

Figure 3.10: Main routine, **function** *update*, and **function** *check_end_point* of the on-line algorithm to construct CDAWGs.

```
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 edges $(s,(k',k'+p-k),r)$ 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 $k' \leq p$ then return $(s',k')$;    /* implicit case. */
 3 /* explicit case. */
 4 if $length(s') = length(s) + (p-k+1)$ then return $(s',k')$;    /* solid case. */
 5 /* non-solid case. */
 6 create node $r'$ as a duplication of $s'$;    /* together with the out-going edges of $s'$ */
 7 $suf(r') := suf(s')$;    $suf(s') := r'$;
 8 $length(r') := length(s) + (p-k+1)$;
 9 repeat
10     replace the $w[k]$-edge from $s$ to $s'$ by edge $(s,(k,p),r')$;
11     $(s,k) := canonize(suf(s),(k,p-1))$;
12 until $(s',k') \neq canonize(s,(k,p))$;
13 return $(r',p+1)$;
```

Figure 3.11: 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. 3.4, they are omitted.

Complexity of the Algorithm.

Theorem 3.3 *Assume Σ is a fixed alphabet. For any string $w \in \Sigma^*$, the proposed algorithm constructs $CDAWG'(w)$ on-line and in $O(|w|)$ time, using $O(|w|)$ space.*

Proof. The linearity proof is in a sense the combination of the one of the on-line algorithm for DAWGs [7] and the one of the on-line algorithm for suffix trees [55]. 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 [7]. 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 in [7], 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 number of iterations in the while loop in *canonize*. By concerning 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 in [55]). 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 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. □

3.5 Construction of the CDAWG for a Set of Strings

For a set S of strings $w_1, w_2, \ldots, w_k$, we consider the set $S' = \{w_i\$_i \mid w_i \in S \text{ and } \$_i \notin \mathit{Factor}(S) \text{ for } 1 \leq i \leq |S|\}$. We remark that $\mathit{CDAWG}'(S') = \mathit{CDAWG}(S')$ for any set S of strings.

$\mathit{CDAWG}'(S')$ can be constructed by the same algorithm proposed in the previous section, with a slight modification. We use a global variable e_i for each string in S', where $1 \leq i \leq |S|$, which indicates the ending position of open edges for each string. We treat the set S' like a single sequence $t = w_1\$_1 w_2\$_2 \cdots w_k\$_k$. Whenever we encounter an end-marker $\$_i$, 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. Thereby we have the following.

Theorem 3.4 *Assume Σ is a fixed alphabet. For any set S of strings, the proposed algorithm constructs $\mathit{CDAWG}'(S')$ on-line and in $O(\|S'\|)$ time, using $O(\|S'\|)$ space.*

In Figure 3.12 the construction of $\mathit{CDAWG}'(S')$ is displayed, where $S' = \{\mathtt{cocoa}\$_1, \mathtt{cola}\$_2\}$.

Remark. As a secondary effect of the end-markers $\$_i$, we obtain a good feature on $\mathit{CDAWG}'(S')$. For the set $S = \{\mathtt{cocoa}, \mathtt{cola}\}$, $\mathit{CDAWG}(S)$ is shown in Figure 3.13. One can see there are *three* sink nodes though S contains only *two* strings in it. This is obviously because $[\overset{S}{\overrightarrow{\mathtt{a}}}]^R_S = \{\mathtt{a}\}$. In such case, we cannot readily specify what string(s) in S the string $\mathtt{a}$ is a factor of. However, there is no difficulty to specify it in $\mathit{CDAWG}'(S')$, since there exactly exist $|S'| = |S|$ sink nodes in it (see the right of Figure 3.13).

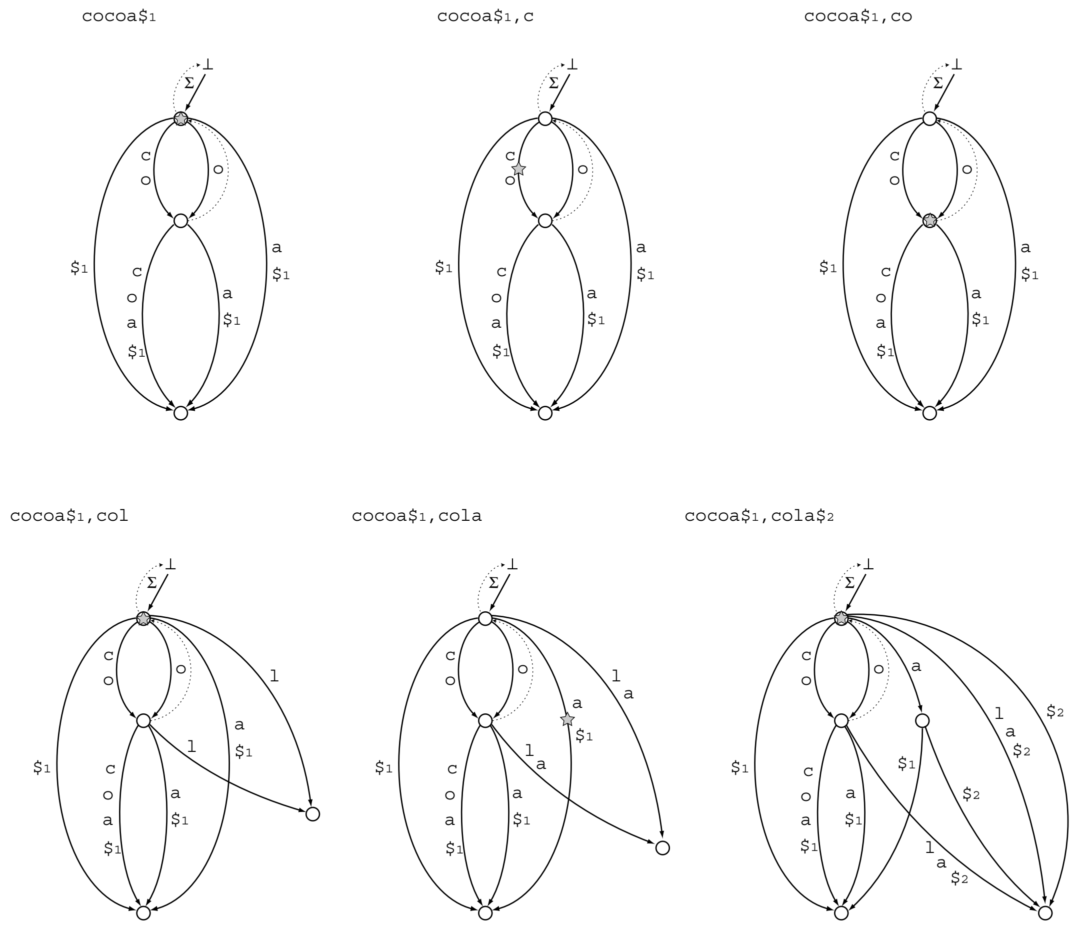


Figure 3.12: Construction of $CDAWG'(S')$ for $S' = \{\texttt{cocoa}\$_1, \texttt{cola}\$_2\}$.

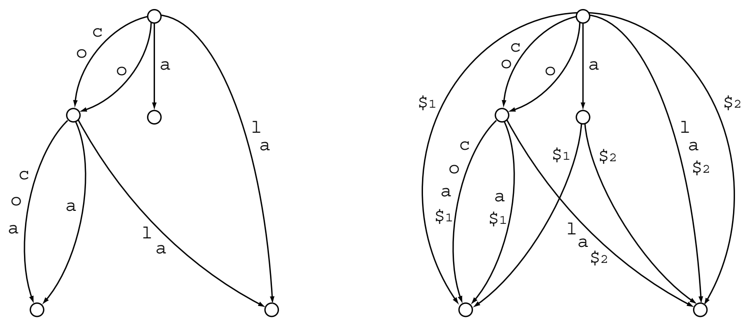


Figure 3.13: For set $S = \{\texttt{cocoa}, \texttt{cola}\}$, $CDAWG(S)$ is shown on the left. For set $S' = \{\texttt{cocoa}\$_1, \texttt{cola}\$_2\}$, $CDAWG'(S')$ is displayed on the right.

Chapter 4

Unification of Algorithms for Constructing Index Structures

4.1 Background and Motivation

When constructing an index structure for a given text string w, what is required primarily is to build it in time *linear* in the length of w. To realize it, much attention and effort has been paid so far [58, 43, 11, 55, 7, 8, 16, 32]. From viewpoints of practice and algorithmics, it is also very important to construct an index structure in *on-line* manner: for example, we can obtain the suffix tree for wa only with small change to append new character a to the existing suffix tree of w. If it is off-line, we have to construct the suffix tree of wa from scratch, even if we beforehand had the suffix tree of w. Therefore, an on-line algorithm constructs an index structure very efficiently, and it allows us to update the input string. Another important factor in constructing an index structure is to build it *for a set of strings* easily. Once constructing index structures for all strings in the set, by merging them it may be possible to obtain an index structure for the set. However, the method is rather straightforward and inefficient, and takes us considerably much time. Hence an algorithm that can *directly* build an index structure for a set of strings is truly helpful.

Table 4.1 shows the on-line algorithms for constructing index structures. Each index structure is suitable to solve particular problems. For example, a suffix tree is optimal to find all the occurrences of a given pattern in a text string [20], a DAWG is a good structure to find the longest common factor of two strings [12], a CDAWG is ideal when we want to save memory space, since its space complexity is strictly smaller than those of the other index structures [8], and so on. Therefore, we should need every index structure in order

Index Structure	*Algorithm*	*linear time*	*on-line*	*multiple strings*
suffix tries	Ukkonen [55]		$\surd$	$\surd$
suffix trees	Ukkonen [55]	$\surd$	$\surd$	$\surd$
DAWGs	Blumer et al. [8]	$\surd$	$\surd$	$\surd$
CDAWGs	Inenaga et al. [32]	$\surd$	$\surd$	$\surd$

Table 4.1: On-line algorithms to construct index structures for a set of strings.

to solve various problems. The matter is, however, that we then have to implement at least four different algorithms, as there exist four index structures.

We had thereby been motivated to get rid of this trouble, and finally succeeded to *unify* the four distinct algorithms, each of which constructs suffix tries, suffix trees, DAWGs, and CDAWGs, respectively. That is, we produce a *generic* algorithm that is capable of constructing any of suffix tries, suffix trees, DAWGs, and CDAWGs. The algorithm is endowed with all the desired properties: it runs on-line and in linear time, and can apply to a set of text strings. However, as an exception the construction of suffix tries cannot always be achieved in linear time since they can require quadratic space.

A complete pseudo-code of our algorithm is shown in Fig. 4.1, Fig. 4.2, Fig. 4.3 and Fig. 4.4. We have marked each line with four symbols: There, $\clubsuit$ ($\spadesuit$, $\heartsuit$ and $\diamondsuit$, resp.) indicates the lines that can be executed when a suffix trie (a suffix tree, a DAWG, and a CDAWG, reps.) is constructed. It has succeeded to reveal the essential common points and separate the small differences among the typical algorithms [55, 7, 32]. In fact, all the control blocks are exactly the same and all differences can be packed into the only *one* procedure in Fig. 4.2 to create a new edge. This means that we can choose which index structure to build, by 'switching' the one procedure.

Furthermore, by comparing the definitions of index structures given in Chapter 2.3 with the algorithm proposed here, some correspondence between them are revealed. Thus, in a sense we provide an algorithmic unified view for the index structures.

The result reported in this chapter was published in [31].

4.2 A Generic Algorithm Constructing Index Structures

Our algorithm to be shown later on constructs $\mathit{STree}'(S)$ and $\mathit{CDAWG}'(S)$ rather than $\mathit{STree}(S)$ and $\mathit{CDAWG}(S)$, since it processes input strings in on-line manner. In the following, we explain the algorithm starting with the common part to every index structure, then we separately give an exposition for the different part.

4.2.1 Construction of an Index Structure for a Single String

We firstly consider the case that the input for the algorithm is a single string $w \in \Sigma^*$. Let $\mathit{Index}(w[1:i])$ be an arbitrary index structure of string $w[1:i]$ for $1 \le i \le |w|$. Our algorithm updates $\mathit{Index}(w[1:i])$ to $\mathit{Index}(w[1:i+1])$ by inserting the suffixes of $w[1:i+1]$ into $\mathit{Index}(w[1:i])$. Recall Definition 3.1 of the LRS, given in Section 3.2. The suffixes of $w[1:i+1]$ can be divided into the following two groups, by the LRS of $w[1:i+1]$.

(1) Suffixes $w[h:i+1]$ for $1 \le h \le j$ where $w[j+1:i+1]$ is the LRS of $w[1:i+1]$.

(2) Suffixes $w[l:i+1]$ for $j+1 \le l \le i+2$.

The group (2) is empty in such case that the LRS of $w[1:i+1] = \varepsilon$, that is, in case $j+1 = i+2$.

Notice that we need not newly insert any suffixes in case (2), simply because they have already been represented in $\mathit{Index}(w[1:i])$. Meanwhile, we insert each suffix of case (1) into $\mathit{Index}(w[1:i])$, from $w[1:i+1]$ to $w[j:i+1]$. We call $w[j+1:i+1]$ *the longest duplicated suffix*, *the LRS* for short, of $w[1:i+1]$. Let us call *the start point* of $\mathit{Index}(w[1:i+1])$ the location where the LRS of $w[1:i]$ is represented in $\mathit{Index}(w[1:i+1])$, and call *the end point* of $\mathit{Index}(w[1:i+1])$ the location where the LRS of $w[1:i+1]$ is represented in $\mathit{Index}(w[1:i+1])$. The suffixes of case (1) can moreover be divided into the following two sub-cases by integer j'.

(1-a) Suffixes $w[h':i+1]$ for $1 \le h' \le j'$ where $w[j'+1:i]$ is the LRS of $w[1:i]$.

(1-b) Suffixes $w[h'':i+1]$ for $j'+1 \le h'' \le j$.

```
♣♠♡♢   Algorithm Construction of an index structure on $\Sigma = \{w[-1], \ldots, w[-m]\}$.
♣♠♡♢    1 create nodes root and $\perp$;
♣♠♡♢    2 for $j := 1$ to $m$ do create a new edge $(\perp, (-j, -j), root)$;
♣♠♡♢    3 $suf(root) := \perp$; $suf(\perp) :=$ nil    /* suffix link */
♣♠♡♢    4 $length(root) := 0$; $length(\perp) := -1$;
♣♠♡♢    5 $(s, k) := (root, 1)$; $i := 0$; $h := 1$; $(t, q) := (root, 1)$;
♣♠♡♢    6 repeat
♣♠♡♢    7     $i := i + 1$;
♣♠♡♢    8     $((s, k), (t, q)) := update((s, k), (t, q), i)$;
♣♠♡♢    9     if $w[i] = endmarker$ then
♣♠♡♢   10         $h := h + 1$;
♣♠♡♢   11         $(t, q) := (root, i + 1)$;
♣♠♡♢   12 until $w[i] =$ EOF;
```

Figure 4.1: Main routine of our algorithm.

The main routine of our new algorithm is shown in Fig. 4.1. In the algorithm, an edge (u, α, v) is represented by $(u, (k, p), v)$ such that k (resp. p) represents the beginning position (resp. the ending position) of the label in the input string w. The main routine calls the **function** *update*, shown in Fig. 4.2, each time a new character is scanned. The **function** *update* plays the main role to update the index structure with a newly scanned character.

In *update* the suffixes of case (1) are inserted, while it is checked whether or not the suffix currently focused on is the LRS of $w[1 : i + 1]$. This is examined by the **function** *check_end_point*, in the 2nd line of *update*. In the 12th line a new edge is created for each of the suffixes in case (1), and the way to do it depends on which index structure we are constructing, as shown in the lower part of Fig. 4.2. The detail of the dependence will be mentioned in the sequel. The location from which the algorithm should insert each suffix in case (1) is called *the active point* for the suffix. Where the active point should start on updating the structure also depends on which index structure we are constructing.

Now suppose that we have just before finished inserting a suffix $w[h : i + 1]$ where $j' + 1 \leq h \leq j - 1$, which is in case (1-a). Then, in the 15th line of *update* the active point is moved to the location where the string $w[h + 1 : i + 1]$ is associated, via the suffix link. The reference pair for $w[h + 1 : i + 1]$ is then canonized by the **function** *canonize*. This operation is continued until the LRS of $w[1 : i + 1]$, i.e. the end point, is found.

```
♣♠♡♢     **function** $update((s,k),(t,q),p)$: pairs of node and **integer**;
♣♠♡♢     /* $(t,(q,p-1))$ *is the canonical reference pair for the advanced point.* */
♣♠♡♢      1 $c := w[p]$; $oldr :=$ **nil**; $s' :=$ **nil**;
♣♠♡♢      2 **while not** $check\_end\_point(s,(k,p-1),c)$ **do**
♣♠♡♢      3     **if** $k \le p-1$ **then**    /* *implicit* */
 ♠ ♢      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** $r := s$;    /* *explicit* */
♣♠♡♢     12     $\boxed{\textbf{CreateNewEdge}}$    /* *Change only this line* */
♣♠♡♢     13     **if** $oldr \neq$ **nil then** $suf(oldr) := r$;
♣♠♡♢     14     $oldr := r$;
♣♠♡♢     15     $(s,k) := canonize(suf(s),(k,p-1))$;
♣♠♡♢     16 **if** $oldr \neq$ **nil then** $suf(oldr) := s$;
♣♠♡♢     17 $(s,k) := separate\_node(s,(k,p))$;
♣♠♡♢     18 $(t,q) := canonize(t,(q,p))$;
♣♠♡♢     19 **if** $q > p$ **then** $(s,k) := (t,q)$;    /* *the advanced point is explicit* */
♣♠♡♢     20 **return** $((s,k),(t,q))$;
```

$\boxed{\textbf{CreateNewEdge}}$ should be replaced as follows respectively.

For Suffix Tries

```
create a new node $v$;
$length(v) := length(r) + 1$;
create a new edge $(r,(p,p),v)$;
```

For Suffix Trees

```
create a new node $v$;
$length(v) := \infty$;
create a new edge $(r,(p,\infty),v)$;
```

For DAWGs

```
*if* $v$ *has not been defined yet*
    create a new node $v$;
    $length(v) := length(r) + 1$;
create a new edge $(r,(p,p),v)$;
```

For CDAWGs

```
*if* $s_h$ *has not been defined yet*
    create a sink node $s_h$;    /* $s_h$ *is a global variable* */
    $length(s_h) := \infty$;
create a new edge $(r,(p,\infty),s_h)$;
```

Figure 4.2: Function *update*.

```
♣♠♡♢    function $check\_end\_point(s,(k,p),c)$: boolean;
♣♠♡♢     1 if $k \le p$ then    /* implicit */
 ♠ ♢     2     let $(s,(k',p'),s')$ be the $w[k]$-edge from $s$;
 ♠ ♢     3     return $(c = w[k'+p-k+1])$;
♣♠♡♢     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 $w[k]$-edge $(s,(k',p'),s')$ from $s$; return $s'$;    /* implicit */

   ♢    function $redirect\_edge(s,(k,p),r)$;
   ♢     1 let $(s,(k',p'),s')$ be the $w[k]$-edge from $s$;
   ♢     2 replace this edge by edge $(s,(k',k'+p-k),r)$;

♣♠♡♢    function $canonize(s,(k,p))$: pair of integers;
♣♠♡♢     1 if $k > p$ then return $(s,k)$;    /* explicit */
 ♠ ♢       /* $(s,(k,p))$ is an implicit node. */
 ♠ ♢     2 find the $w[k]$-edge $(s,(k',p'),s')$ from $s$;
 ♠ ♢     3 while $p'-k' \le p-k$ do
 ♠ ♢     4     $k := k+p'-k'+1$; $s := s'$;
 ♠ ♢     5     if $k \le p$ then find the $w[k]$-edge $(s,(k',p'),s')$ from $s$;
 ♠ ♢     6 return $(s,k)$;
```

Figure 4.3: Functions *check_end_point*, *extension*, and *canonize*.

```
♠ ♢    function $split\_edge(s,(k,p))$: node;
♠ ♢     1 let $(s,(k',p'),s')$ be the $w[k]$-edge from $s$;
♠ ♢     2 replace the 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 $r$;

♣♠♡♢   function $separate\_node(s,(k,p))$: pair of node integer;
♣♠♡♢    1 $(s',k') := canonize(s,(k,p))$;
♣♠♡♢    2 if $k' \le p$ then return $(s',k')$;    /* implicit */
♣♠♡♢      /* $(s',(k',p))$ is an explicit node. */
♣♠♡♢    3 if $length(s') = length(s)+p-k+1$ then return $(s',k')$;    /* solid */
♠ ♢       /* non-solid case */
♠ ♢     4 create a new node $r'$ as a duplication of $s'$;
♠ ♢     5 $suf(r') := suf(s')$; $suf(s') := r'$;
♠ ♢     6 $length(r') := length(s)+(p-k+1)$;
♠ ♢     7 repeat
♠ ♢     8     replace the $w[k]$-edge from $s$ to $s'$ by edge $(s,(k,p),r')$;
♠ ♢     9     $(s,k) := canonize(suf(s),(k,p-1))$;
♠ ♢    10 until $(s',k') \ne canonize(s,(k,p))$;
♠ ♢    11 return $(r',p+1)$;
```

Figure 4.4: Other functions.

What we mentioned above are common to all the four index structures. From now on, let us treat the differences, $\boxed{\textbf{CreateNewEdge}}$.

4.2.2 $\boxed{\textbf{CreateNewEdge}}$ in Case of Suffix Tries

Assume that we now have $STrie(w[1:i])$. First, we insert the suffixes of case (1-a) into the suffix trie. Definition 2.7 tells that every edge of a suffix trie must be labeled with a single character. Therefore, from each leaf node of $STrie(w[1:i])$ a new edge labeled by $w[i+1:i+1]$ is created together with a new leaf node. This way the suffixes of case (1-a) are inserted. The update from $STrie(w[1:i])$ to $STrie(w[1:i+1])$ should begin at the node $w[1:i]$. We call this location *the advanced point* of $STrie(w[1:i])$. The active point was reset to node $w[1:i]$ after the construction of $STrie(w[1:i])$ had been finished, and this was done in the 19th line of *update*. Second, we insert the suffixes of case (1-b). By creating new edges labeled by $w[i+1:i+1]$ from nodes $w[h:i]$ where $j'+1 \le h \le j$, they are inserted.

4.2.3 $\boxed{\textbf{CreateNewEdge}}$ in Case of Suffix Trees

Assume that we now have $STree'(w[1:i])$. Recall Definition 2.8. A careful consideration reveals the fact that any leaf node of $STree'(w[1:i])$ will also be a leaf node in $STree'(w[1:k])$ for any k where $i+1 \leq k \leq |w|$. Hence we refer the second value of label of an edge directing a leaf node of $STree'(w[1:i])$ to "∞" as Ukkonen did in [55], and do the length of the leaf node to "∞" as well. This way is so ingenious that we need not explicitly insert any suffix in case (1-a). In addition, it inherently corresponds to the "compaction" from a suffix trie to a suffix tree, shown in Fig. 3.1. Then all we have to do is to insert the suffixes of case (1-b) into the suffix tree. That is why the active point should be on the start point of $STree'(w[1:i])$ at the beginning of the update. Consider the case that the active point is on an edge (on an implicit node) and corresponds to string $w[h:i]$ for some $j'+1 \leq h \leq j$. Since $w[h:i+1]$ is not the LRS of $w[h:i+1]$, naturally it has to be inserted into the suffix tree. To do it, a new node is created where the active point is. In other words, the implicit node becomes explicit. This is done by the **function** *split_edge* called in the 10th line of *update*.

4.2.4 $\boxed{\textbf{CreateNewEdge}}$ in Case of DAWGs

Assume that we now have $DAWG(w[1:i])$. Definition 2.9 tells that only strings ending at the same position in w must be represented in the same node. String $w[h:i+1]$ belongs to $[w[1:i+1]]^R_{w[1:i+1]}$ for any h with $1 \leq h \leq j'$, which is a suffix in case (1-a). These result in the fact that an edge labeled by $w[i+1:i+1]$ and the new sink node should be created from the last sink node $[w[1:i]]^R_{w[1:i+1]}$, and by this procedure all the suffixes of case (1-a) are inserted. Therefore, as in case of suffix tries, in the 19th line of *update* the active point was moved to the advanced point of $DAWG(w[1:i])$ after its construction had been completed. To insert a suffix $w[h:i+1]$ in case (1-b) for $j'+1 \leq h \leq j$, a new edge labeled with $w[i+1:i+1]$ is created from node $[w[h:i]]^R_{w[1:i+1]}$ to the new sink node $[w[1:i+1]]^R_{w[1:i+1]}$. It corresponds to the "minimization" from a suffix trie to a DAWG. Suppose that the end point has already found, that is, the insertion of all the suffixes of $w[1:i+1]$ has been finished, and focus on the LRS $w[j+1:i+1]$ of $w[1:i+1]$. In the 17th line of *update* the **function** *separate_node* is called, which examines whether or not $w[j+1:i+1] = u$, where $u = \overleftarrow{w[j+1:i+1]}^{w[1:i+1]}$. If not, $w[j+1:i+1]$ cannot any longer be represented in the same node as u. Therefore, the node is separated into two nodes,

$[u]^R_{w[1:i+1]}$ and $[w[j+1:i+1]]^R_{w[1:i+1]}$.

4.2.5 $\boxed{\text{CreateNewEdge}}$ in Case of CDAWGs

Assume that we now have $CDAWG'(w[1:i])$. The way to update $CDAWG'(w[1:i])$ is like combination of those to update $STree'(w[1:i])$ and $DAWG(w[1:i])$. Let us call *the terminal edges* the edges directing the sink node of a CDAWG. It follows from Definition 2.10 that any terminal edge of $CDAWG'(w[1:i])$ will also be a terminal edge of $CDAWG'(w[1:k])$ for any k where $i+1 \leq k \leq |w|$. Hence we refer the second value of any terminal edge to "∞", like the case of suffix trees. This way every suffix of case (1-a) is implicitly inserted to the CDAWG, therefore the active point starts at the start point of $CDAWG'(w[1:i])$. Suppose the active point is on an edge (on an implicit node) right before inserting $w[j'+1:i+1]$. Then the edge is split into two, due to the creation of the node from which an edge with label $w[i+1:\infty]$ is created. This way to label the edge corresponds to the "compaction" from a DAWG to a CDAWG. The edge is directed to the sink node. It corresponds to the "minimization" from a suffix tree to a CDAWG. To insert $w[j'+2:i+1]$, the active point is moved to the location with which $w[j'+2:i]$ is associated. If it is an implicit node, in the 4th line of *update* it is examined if $w[j'+2:i+1]$ is to belong to $[w[j'+1:i]]^R_{w[1:i+1]}$. If so, the edge is redirected to the node last created, $[w[j'+1:i]]^R_{w[1:i+1]}$, and its label is modified accordingly. The **function** *redirect_edge* accomplishes the operation above. If not, a new node for $[w[j'+2:i]]^R_{w[1:i]}$ is newly created, so that a new edge labeled with $w[j+1:\infty]$ can be created from it to the sink node.

4.2.6 Extension to a Set of Strings

Given a set $S = \{w_1, w_2, \ldots, w_k\}$, we regard it as a sequence $t = w_1\$_1 w_2\$_2 \cdots w_k\$_k$, where $\$_h$ is the end-marker of w_h for $1 \leq h \leq k$. This way we can treat S like one string. In the 9th line of the main routine in Fig. 4.1, if $t[i]$ is an end-marker, integer h counting the number of the input strings is increased one, and the advanced point is reset to the root node preparing for the next string. The active point is also to be on the root node, since any end-marker never appears in any string in S. Consequently, the algorithm builds $STrie(S)$, $STree'(S)$, $DAWG(S)$, and $CDAWG'(S)$, for a given set S of strings.

As a result of the discussion, we have the following.

Theorem 4.1 *For any set of strings the proposed algorithm constructs $STrie(S)$, $STree'(S)$, $DAWG(S)$, and $CDAWG'(S)$ on-line, and in liner time except for $STrie(S)$, by changing the 12th line of the* **function** *update accordingly.*

For comparison, for $S = \{\texttt{abab}, \texttt{abb}\}$, the on-line constructions of $STrie(S)$, $STree'(S)$, $DAWG(S)$, and $CDAWG'(S)$ are shown in respectively Fig. 4.5, Fig. 4.6, Fig. 4.7, and Fig. 4.8.

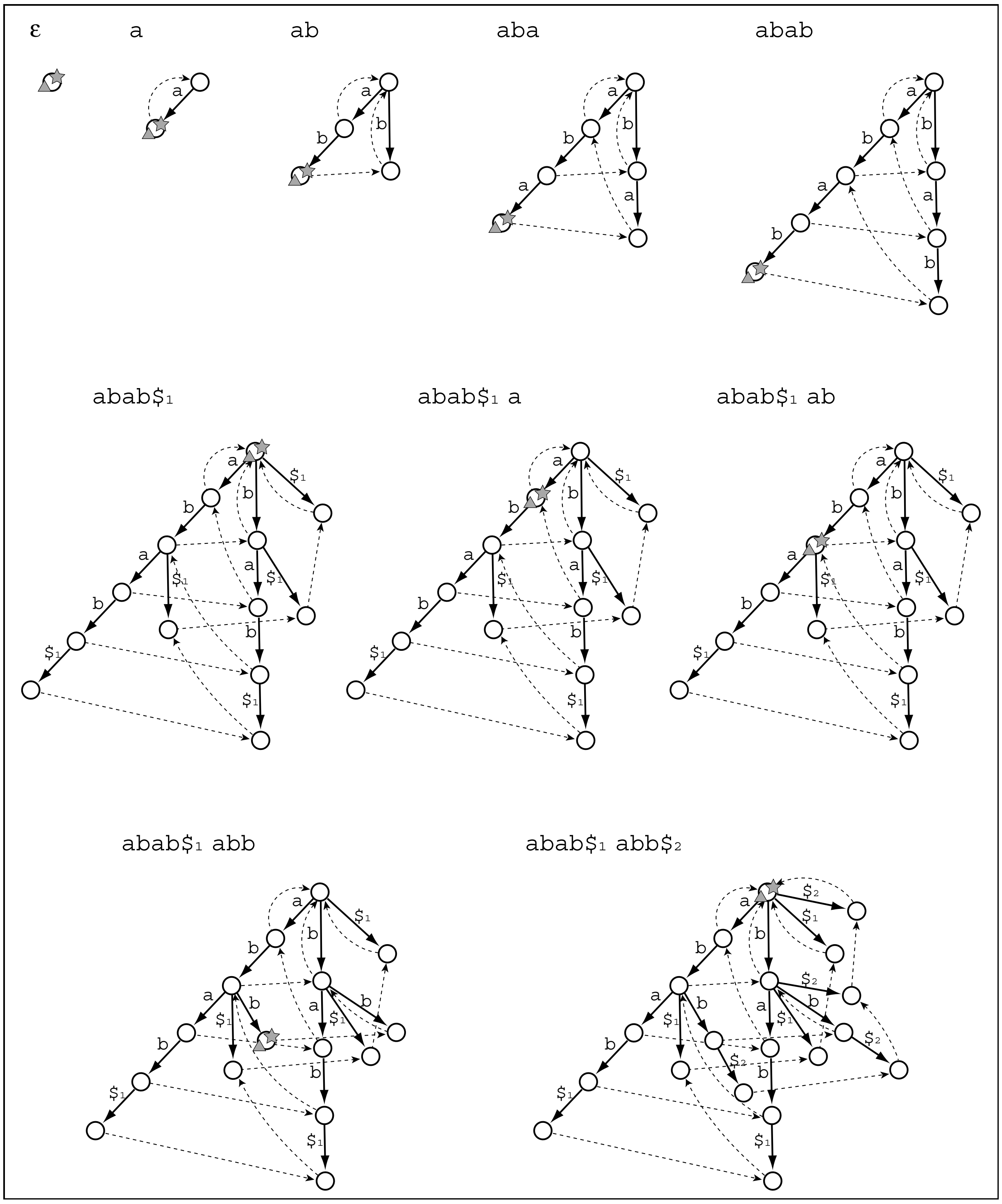


Figure 4.5: A snapshot of the on-line construction of $STrie(S)$ where $S = \{\texttt{abab}, \texttt{abb}\}$. The gray star and the triangle represent the active point and the advanced point of each step, respectively. The broken lines are suffix links.

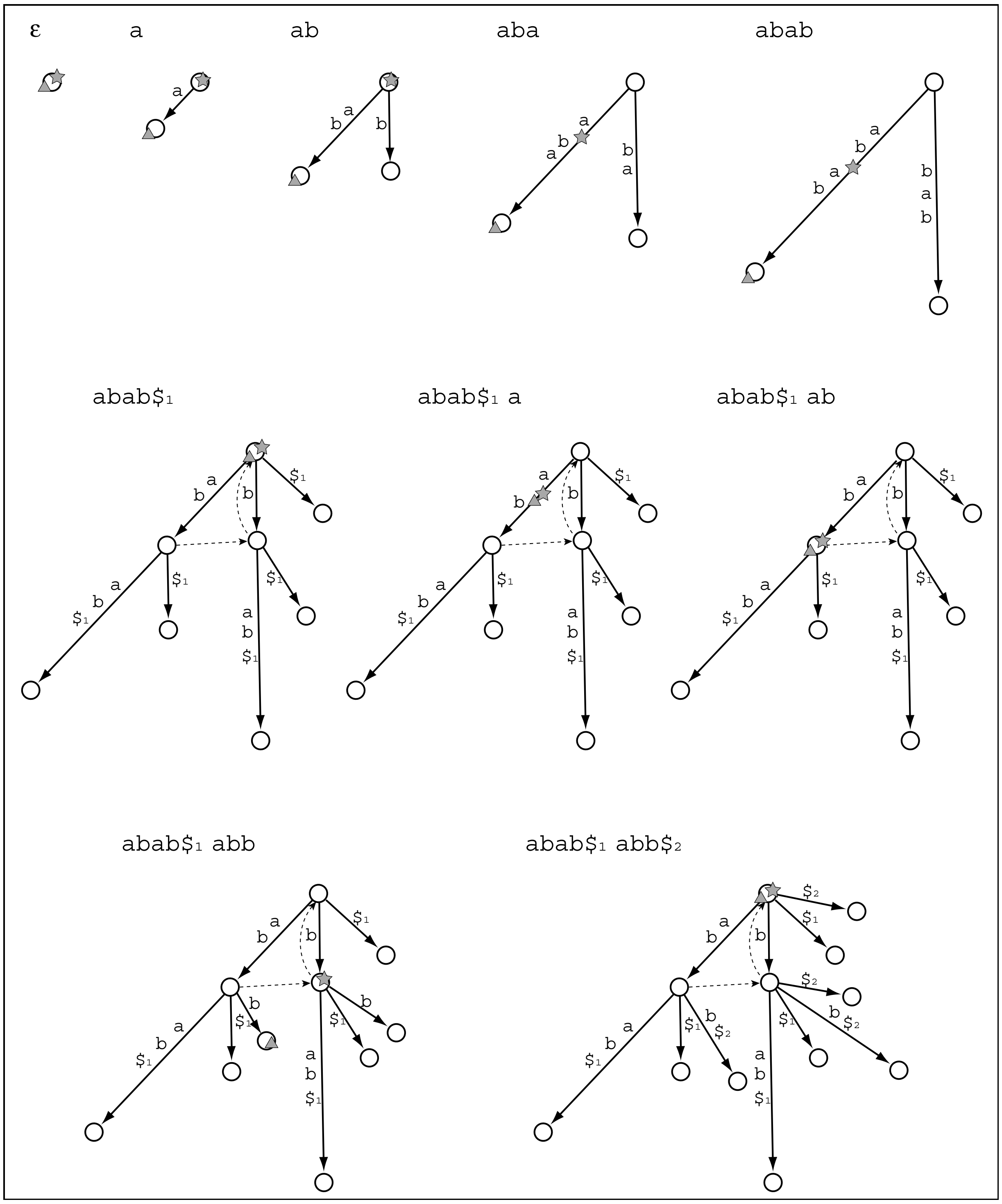


Figure 4.6: A snapshot of the on-line construction of $STree'(S)$ where $S = \{\mathtt{abab}, \mathtt{abb}\}$. The gray star and the triangle represent the active point and the advanced point of each step, respectively. The broken line are suffix links.

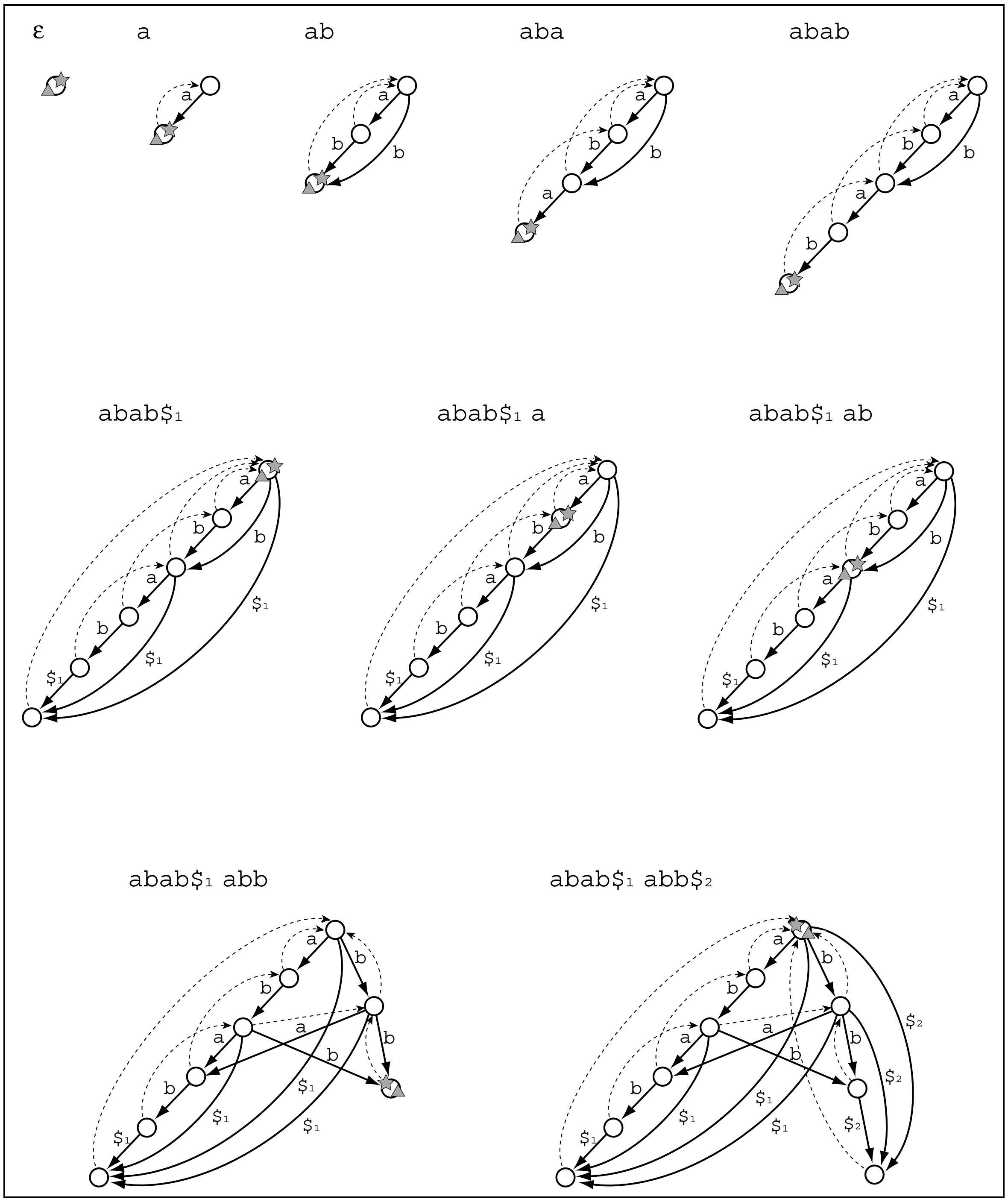


Figure 4.7: A snapshot of the on-line construction of $DAWG(S)$ where $S = \{\mathsf{abab}, \mathsf{abb}\}$. The gray star and the triangle represent the active point and the advanced point of each step, respectively. The broken lines are suffix links.

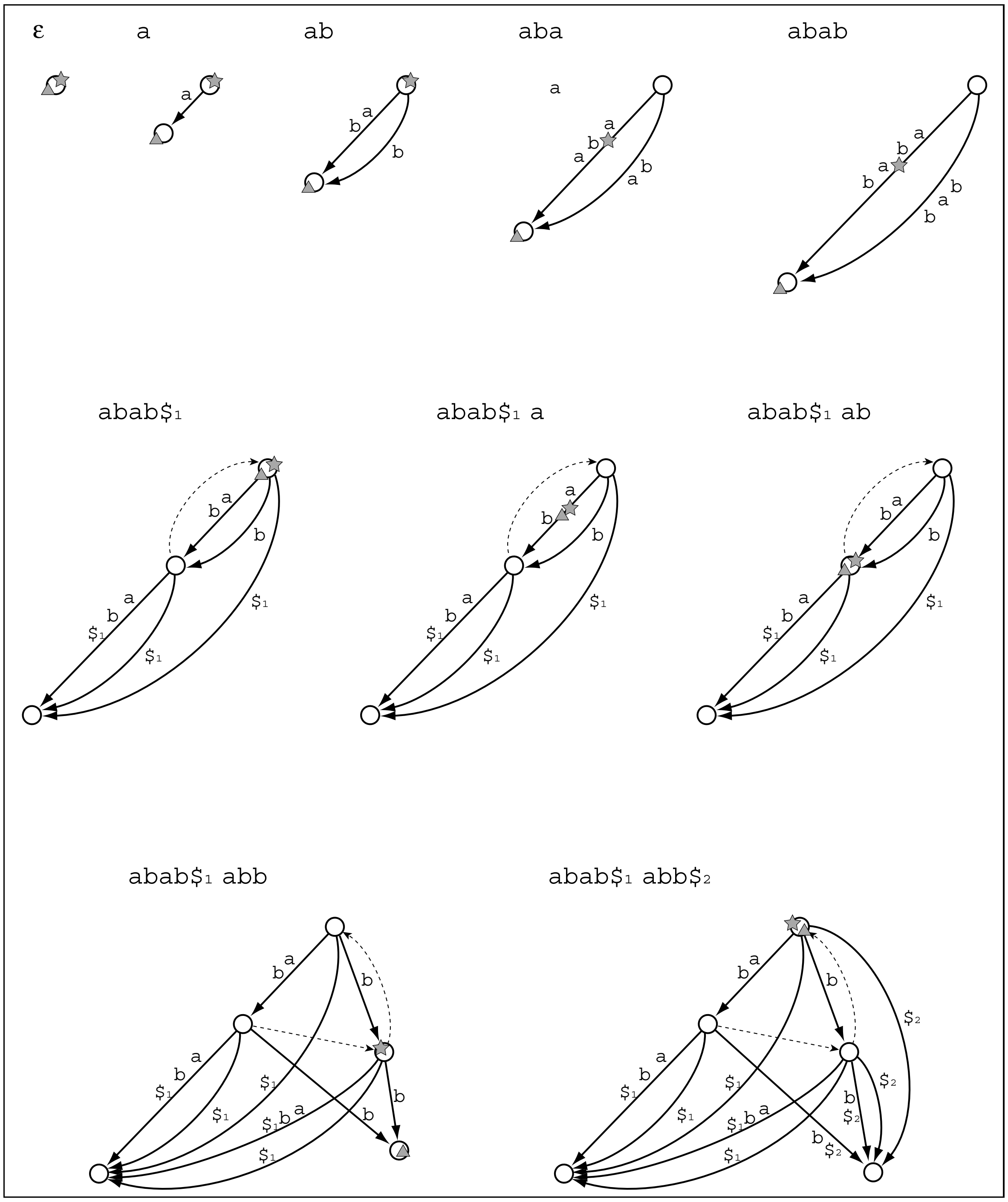


Figure 4.8: A snapshot of the on-line construction of $CDAWG'(S)$ where $S = \{\mathtt{abab}, \mathtt{abb}\}$. The gray star and the triangle represent the active point and the advanced point of each step, respectively. The broken lines are suffix links.

Chapter 5

Construction of the CDAWG for a Trie

5.1 Introduction

In Chapter 3 and Chapter 4, we have discussed the construction of the CDAWG for a set S of strings. In this chapter, we consider the case that the set S is given in the form of a trie (see Section 2.1.1). Namely, our input is $\mathit{Trie}(S)$ and output is $\mathit{CDAWG}'(S)$. Let $\|S\| = \ell$. Since the trie shares common prefixes of the strings in S, in general the number N of nodes of the trie is less than ℓ. We show a non-trivial extension of the algorithm that constructs CDAWG for a trie in $O(N)$ time and space. The algorithm is designed on the basis of the one for constructing CDAWGs for a set of strings, which has been introduced in Chapter 3.

Some related work can be seen in literature: Kosaraju [37] introduced the suffix tree for a *reversed* trie. We denote it by $\mathit{Trie}^{\mathrm{rev}}(S)$. Let M be the number of nodes in $\mathit{Trie}^{\mathrm{rev}}(S)$. Kosaraju showed an algorithm to construct $\mathit{STree}(S)$ in $O(M \log M)$ time. Later on, Breslauer [9] reduced it to $O(M)$ time.

On the other hand, our algorithm constructs a CDAWG for a (normal) trie. We believe our assumption that a set S of strings is given as $\mathit{Trie}(S)$ is natural. In addition, we remark that the algorithm to be proposed also becomes capable of building $\mathit{STree}(S)$ and $\mathit{DAWG}(S)$ in $O(N)$ time, in the combination with the generic algorithm introduced in Chapter 4. $\mathit{STrie}(S)$ can also be constructed in $O(N^2)$ time.

The result involved in this chapter was published in [29].

5.2 Trie and Reversed Trie

We define the *reversed trie* for a set S of strings as a *reverse-directed* tree. We denote it by $\mathit{Trie}^{\mathrm{rev}}(S)$. The root node of $\mathit{Trie}^{\mathrm{rev}}(S)$ is out-degree zero and any leaf node is in-degree zero. Formally, $\mathit{Trie}^{\mathrm{rev}}(S)$ is defined as follows.

Definition 5.1 *$\mathit{Trie}^{\mathrm{rev}}(S)$ is the tree (V, E) such that*

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

We define a counterpart of the prefix property for a set of strings.

Definition 5.2 *Let $S = \{w_1, \dots, w_k\}$ where $w_i \in \Sigma^*$ for $1 \le i \le k$ and $k \ge 1$. We say that S has the* suffix property *iff $w_i \notin \mathit{Suffix}(w_j)$ for any $1 \le i \ne j \le k$.*

Then, the following obvious proposition holds.

Proposition 5.1 *Any string in $\mathit{Prefix}(S)$ can be spelled out from a leaf node in $\mathit{Trie}^{\mathrm{rev}}(S)$ iff a set S of strings has the suffix property.*

It directly follows from the contraposition of the above proposition that, if S does not have the suffix property, we cannot spell out every string in $\mathit{Prefix}(S)$. Since Breslauer's algorithm [9] traverses a given reversed trie from a leaf node, it is not supposed to construct the suffix tree for a set of strings that does not have the suffix property.

Proposition 5.2 *Given a set $S = \{w_1, \dots, w_k\}$ such that $w_i \notin \mathit{Suffix}(w_j)$ for any $1 \le i \ne j \le k$, let $S'' = \{w_1\$, \dots, w_k\$\}$. Then, $\mathit{Trie}^{\mathrm{rev}}(S'')$ has at most $\|S''\| - |S''| + 2 = \|S\| + 2$ nodes.*

The input of Breslauer's algorithm is $\mathit{Trie}^{\mathrm{rev}}(S'')$. If the strings in S'' have long and many common suffixes, the number of nodes in $\mathit{Trie}^{\mathrm{rev}}(S'')$ is by far smaller than the upper bound $\|S''\| - |S''| + 2$.

$\mathit{Trie}^{\mathrm{rev}}(S'')$ for $S'' = \{aaab\$, aac\$, aa\$, abc\$, bab\$, ba\$\}$ is shown in Fig. 5.1.

Theorem 5.1 (Breslauer [9]) *Breslauer's algorithm constructs the suffix tree of a reversed trie in time proportional to the number of nodes in the reversed trie.*

On the other hand, given a set $S = \{w_1, \dots, w_k\}$ with $k \ge 1$, we consider the set $S' = \{w_1\$_1, \dots, w_k\$_k\}$ where $\$_i$ denotes the unique end-marker for w_i $(1 \le i \le k)$.

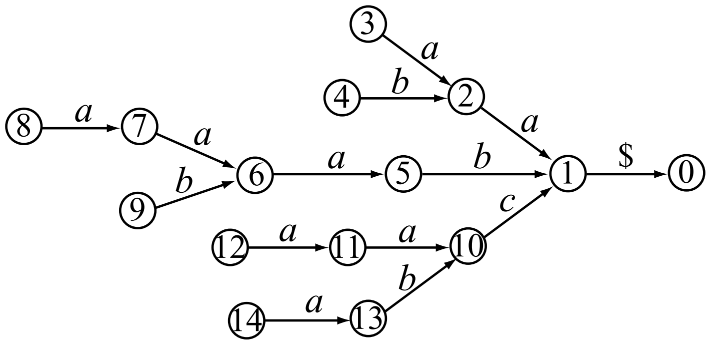


Figure 5.1: *Trie*$^{\text{rev}}(S'')$ for $S'' = \{aaab\$, aac\$, aa\$, abc\$, bab\$, ba\$\}$.

Proposition 5.3 *Given a set $S = \{w_1, \ldots, w_k\}$ with $k \geq 1$, let $S' = \{w_1\$_1, \ldots, w_k\$_k\}$. Then, Trie$(S')$ has at most $\|S'\| + 1 = \|S\| + |S| + 1$ nodes.*

See Fig. 5.2 in which *Trie*(S') is displayed, where $S' = \{aaab\$_1, aac\$_2, aa\$_3, abc\$_4, bab\$_5, ba\$_6\}$. Even if set S does not have the prefix property, every string $x \in S'$ corresponds to a leaf node. In fact, although a string aa is a prefix of a string $aaab$, the path spelling out $aa\$_3$ ends at leaf node 8 in Fig. 5.2.

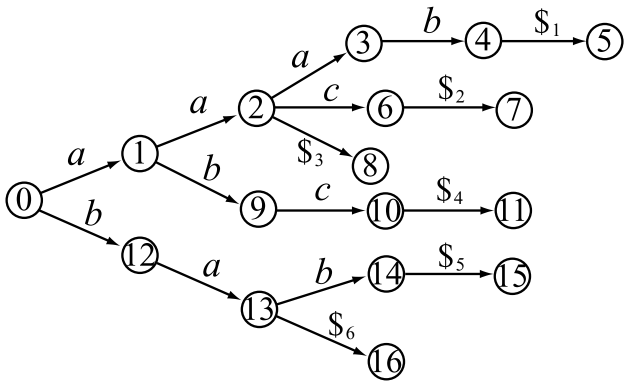


Figure 5.2: *Trie*(S') for $S' = \{aaab\$_1, aac\$_2, aa\$_3, abc\$_4, bab\$_5, ba\$_6\}$.

Trie(S') is the input of our algorithm to be introduced in Section 5.3.

5.3 Algorithm to Construct the CDAWG for a Trie

We firstly note that the CDAWG for *Trie*(S') is the same as the CDAWG of S for any set S of string, except the following point. The label of an edge in *CDAWG*(S') is implemented by a triple of integers (h, i, j) representing the starting position i and ending position j of the label in the h-th string in S. Meanwhile, that in the CDAWG for *Trie*(S') refers to a pair of nodes in *Trie*(S'), between which the string corresponding to the label is lying.

The basic action of the algorithm is to update the CDAWG incrementally, synchronized with the depth-first traversal on *Trie*(S'). The key idea to achieve the linear time construction is as follows.

(1) Keep track of the *advanced point* q in the CDAWG so that the path from the root node to q coincides with the path from the root node to node v, where v is the node currently visited in the trie.

(2) Create a new node in the CDAWG where the advanced point q is, before stepping into the first branch at each branching node in the trie.

We will explain the detail in the sequel. Suppose that, after having traveled nodes with scanning $\alpha \in$ *Prefix*(S') in *Trie*(S'), the algorithm encounters a node v having k (≥ 2) branches in *Trie*(S'). Moreover suppose that it then chooses an edge from which to a leaf node a string $\beta \in$ *Suffix*(S') is spelled out. After updating the CDAWG with string $\alpha\beta$, the algorithm has to update it with the other strings represented in *Trie*(S'). Notice that the current CDAWG already has the path representing α from the source node, which corresponds to prefixes of at least k strings in S. Thus the algorithm has to restart updating the CDAWG from the location to which α corresponds, and has to continue traversing *Trie*(S') from the node v. For that purpose, we trace the *advanced point* q mentioned in (1) above.

Let us now clarify the aim of (2). The aim is to make the advanced point q be an *explicit* node whenever the algorithm encounters a branching node in *Trie*(S'). That is, the reference pair of q should then become of the form (s, ε) for some node s. What is the matter if the advanced point q is not explicit before stepping into the first branch? Assume that the advanced point q was referred to as (u, γ) with some node u and string $\gamma \neq \varepsilon$ when the algorithm encountered the node v corresponding to α in *Trie*(S'). After finishing updating the CDAWG with $\alpha\beta$, the algorithm focuses back on v and $q = (u, \gamma)$.

The matter is that the reference (u, γ) might not be *canonical* any longer: the path spelling out γ may contain extra nodes. Namely, the path spelling out γ may have been split while the algorithm updated the CDAWG with string β. A concrete example is shown in Fig. 5.3.

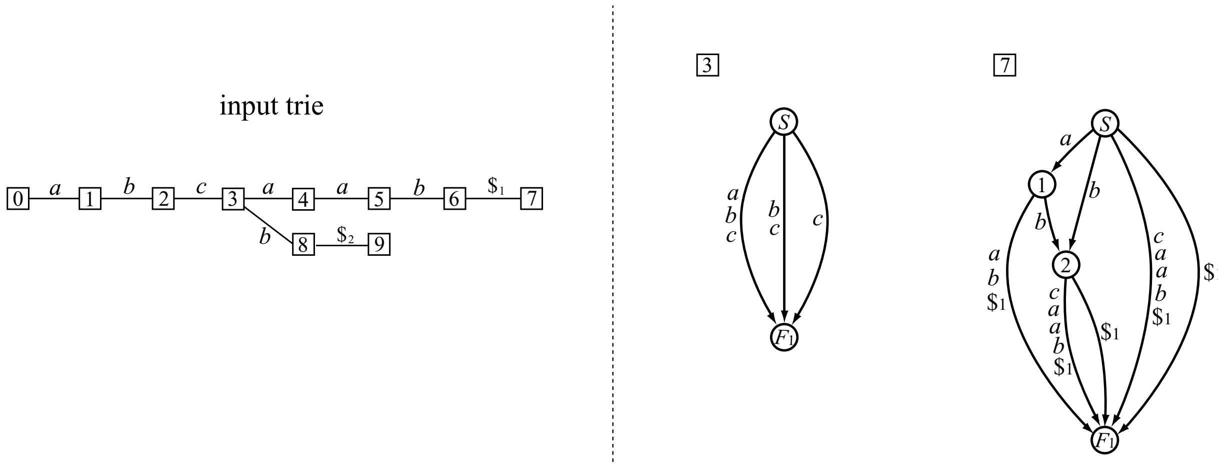


Figure 5.3: *Trie*(S') for $S = \{abcaab\$_1, abcb\$_2\}$ is shown left. When the algorithm focuses on node $\boxed{3}$ in *Trie*(S'), it needs to memorize the location in the CDAWG corresponding to *abc*. Since there is no node but F_1 at the location, it is memorized by a reference pair $(0, abc)$. After having visited node $\boxed{7}$ in *Trie*(S'), the algorithm updates the CDAWG from $(0, abc)$, and with node $\boxed{3}$ in the trie. However, since the path spelling *abc* dose not consist of an edge any more, the algorithm has to find the nearest node from the location the path ends on, that is, node 2. We have to avoid this, because traversing the path spelling *abc* in the CDAWG just deserves traversing *Trie*(S') from node $\boxed{0}$ to $\boxed{3}$.

If the algorithm scans such extra nodes, its time complexity can become quadratic with respect to the number of nodes in *Trie*(S'). In order to avoid this matter, the algorithm creates a new node s so that the active point is guaranteed to be on an explicit node. However, the algorithm dose not merge any other edges because at the moment it is unknown how many edges should be merged into the new node s. Of course, if $\gamma = \varepsilon$, there is no need to create any new node.

The algorithm is described is Fig, 5.4. The variable *current_node* indicates the node on which the algorithm currently focuses in *Trie*(S'). The variable *advanced_point* is of the

```
Algorithm to construct the CDAWG for Trie(S')    /* The input is Trie(S') */
 1 current_node := root;    /* the root node of Trie(S') */
 2 active_point := (source, ε);
 3 advanced_point := (source, ε);
 4 traverse_and_update(current_node, active_point, advanced_point);

procedure traverse_and_update(current_node, active_point, advanced_point)
 1 Let label_set be the set of labels of the outgoing edges of current_node;
 2 if |label_set| = 0 then return;
 3 else if |label_set| ≥ 2 then create_node(advanced_point);
 4 for each c ∈ label_set do
 5     new_active_point := update_CDAWG(c, active_point);
 6     Let new_advanced_point be the location where active_point advances with c;
 7     Let v be the node to which the edge labeled c points;
 8     traverse_and_update(v, new_active_point, new_advanced_point);
```

Figure 5.4: Algorithm to construct the CDAWG for a trie.

form of a reference pair (u, β), where u is the parent node nearest to *advanced_point*. As mentioned above, the string β is actually implemented by a pair of nodes in *Trie*(S'). In the procedure *traverse_and_update*, the **function** *update_CDAWG* updates the CDAWG with a letter c. The **function** *update_CDAWG* is the same as the one for the construction of the CDAWG for a set of strings, introduced in Chapter 3, excepting that *update_CDAWG* creates a new edge stemming from the node latest created by **function** *create_node*.

An example of the construction of the CDAWG for a trie is shown in Fig. 5.3.

Finally, we have the following theorem.

Theorem 5.2 *The proposed algorithm constructs the CDAWG for a trie in linear time and space with respect to the number of nodes in the trie.*

Proof. We first explain that the modification of the **function** *update_CDAWG* and the **function** *create_node* itself do not affect the linearity of the algorithm.

Suppose that an input trie has n nodes. It is clear that the number of nodes visited by *advanced_point* in the CDAWG is at most n. Hence it takes $O(n)$ time to calculate *advanced_point* all through the construction. Furthermore suppose that m nodes in *Trie*(S') are branching. It is clear that $m < n$, because any trie has at least one leaf node.

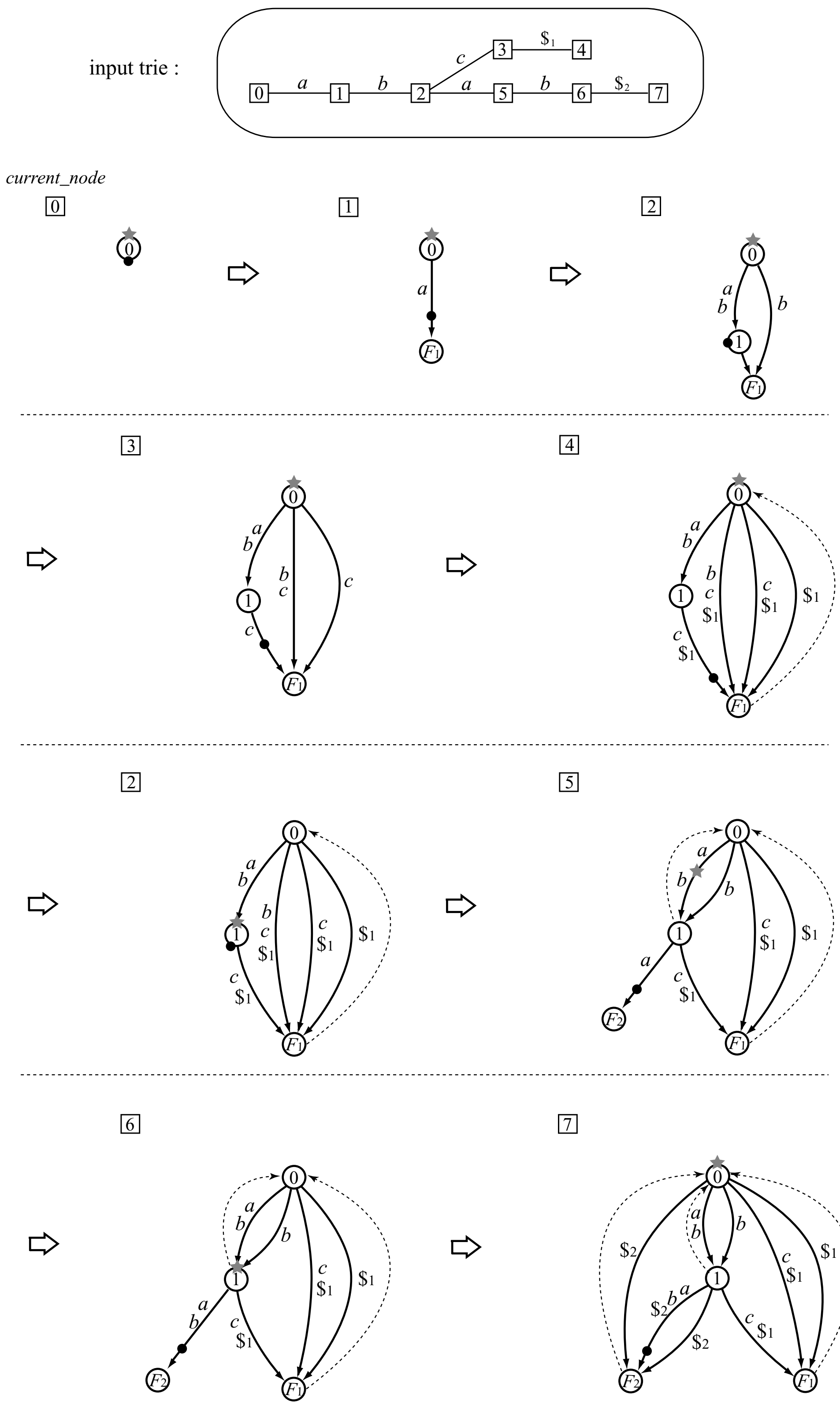


Figure 5.5: Construction of the CDAWG for *Trie*(S'), where $S = \{abc\$_1, abab\$_2\}$. The gray starred point represents *active_point*, and the black dotted point represents *advanced_point*. For simplicity, the bottom node is omitted. As node $\boxed{2}$ in the trie is branching, a new node $\boxed{1}$ is created in the CDAWG when *current_node* arrives at node 2 for the first time. After *current_node* visits node 4, the algorithm updates the CDAWG with *current_node* = 2 and *advanced_point* = 1.

Therefore, the **function** *create_node* creates at most m nodes in the CDAWG, and it implies that the time complexity of *create_node* is $O(m)$. This implies the modification, creating new edges due to the nodes made by the **function** *create_node*, takes $O(m)$ time as well.

We from now on verify the overall linearity of the proposed algorithm. The matter we have to clarify is the upper bound of the number of nodes *active_point* visits throughout the construction. Assume that a node v in the trie has k branches and there is a path spelling α between the root node and v. When *current_node* arrives at node v in the trie for the first time, the **function** *create_node* creates a new node u where *advanced_point* is in the CDAWG. Then *active_point* may traverse at most $k|\alpha|$ nodes from p to the initial node via suffix links until finding the location it can stop on. However, $k \leq |\Sigma|$. Therefore, for a trie with n nodes, the number of nodes *active_point* visits throughout the construction is $O(|\Sigma|n)$. Thus, if Σ is a fixed alphabet, the proposed algorithm constructs the CDAWG for a trie in $O(n)$ time and space. □

5.4 Conclusion

We gave an algorithm for constructing the CDAWG for a trie in linear time and space with respect to the number of nodes in the trie. When input strings are given in the form of a trie, the proposed algorithm constructs the CDAWG for the strings faster than the one presented in [32] directly does from a set of the strings, especially when the strings have many common prefixes. As the space complexity of CDAWGs is bounded strictly lower than that of suffix trees, the algorithm presented in this chapter also allows to save memory space.

Chapter 6

On-Line Construction of Symmetric Compact Directed Acyclic Word Graphs

6.1 Introduction

As seen in literature [58, 43, 7, 8, 55, 16, 32] and the previous chapters, *suffix links* are often used, and essential, for efficient constructions of index structures such as suffix tries, suffix trees, DAWGs, and CDAWGs. An interesting fact is that, for any string $w \in \Sigma^*$, the suffix links of $STrie(w)$ form $STrie(w^{\mathrm{rev}})$ [18]. A DAWG also has a similar property, that is, the suffix links of the $DAWG(w)$ compose $STree(w^{\mathrm{rev}})$ [11]. However, this duality is damaged in case of suffix trees. Namely, the suffix links of $STree(w)$ do not form a structure supporting indexes of w^{rev}. However, the set of suffix links of $STree(w)$ corresponds to a subset of the set of edges of $DAWG(w^{\mathrm{rev}})$ [14].

In order to obtain the complete duality on suffix trees, the *affix tree* is developed by Stoye [50, 51]. Affix trees are the modification of suffix trees so that the suffix links of $ATree(w)$ form $ATree(w^{\mathrm{rev}})$ (see Fig. 6.3). Stoye could not prove his on-line algorithm for constructing affix trees runs in linear time, but Maaß [39] later succeeded to improve it so as to run in linear time. Meanwhile, Blumer et al. [8] 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. 6.2, right).

Ukkonen [55] gave intuitive and excellent on-line algorithms for the construction of *STrie*(w) and *STree*(w), as recalled in Chapter 3. 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 [7] 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 [58], 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 [16]. It implies that his algorithm also simultaneously constructs $DAWG(w)$ together with *STree*(w^{rev}) on-line, left to right.

In this chapter, we first give an algorithm that simultaneously builds *STree*(w) with $DAWG(w^{\mathrm{rev}})$ on-line, left to right. This algorithm constructs *STree*(w) in the same way as Ukkonen's algorithm does, while computing the *shortest extension links* (sext links) that form $DAWG(w^{\mathrm{rev}})$ 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 introduced in [32], and computes the sext links that are equal to the edges of $CDAWG(w^{\mathrm{rev}})$.

From a practical point of view, SCDAWGs and affix trees have the essentially 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

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

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 considerably save space, compared to affix trees. Moreover, not only an SCDAWG is attractive as index 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 are involved in evaluation/interpretation task for mined expressions from anthologies of classical Japanese poems [52].

The result shown in the chapter was published in [30].

6.2 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 others for w^{rev}.

Giegerich and Kurtz [18] 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 follows.

Definition 6.1 *$STrie(w)$ with $STrie(w^{\mathrm{rev}})$ is the bidirectional tree $(V, E_{L\to R}, E_{R\to L})$ such that*

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

It is obvious that there is a trivial one-to-one correspondence between the set $E_{R\to L}$ and the set F for the suffix links of $STrie(w)$ in Definition 2.7.

The duality of $STree(w)$ and $DAWG(w^{\mathrm{rev}})$, which was pointed out in [11, 14], is shown in Definition 6.2.

Definition 6.2 *$STree(w)$ with $DAWG(w^{\mathrm{rev}})$ is the bidirectional dag $(V, E_{L\to R}, E_{R\to L})$ such that*

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

Let $V' = \{[x]^L_w \mid x \in Factor(w)\}$. It is easy to see that there is a trivial one-to-one correspondence between the node set V of Definition 6.2 and V'. Using this correspondence, we can identify $E_{R\to L}$ of Definition 6.2 with

$$\begin{array}{cl} & \{([x]^L_w, a, [ax]^L_w) \mid x, ax \in Factor(w) \text{ and } a \in \Sigma\} \\ = & \{([y]^R_{w^{\mathrm{rev}}}, a, [ya]^R_{w^{\mathrm{rev}}}) \mid y, ya \in Factor(w^{\mathrm{rev}}) \text{ and } a \in \Sigma\}, \end{array}$$

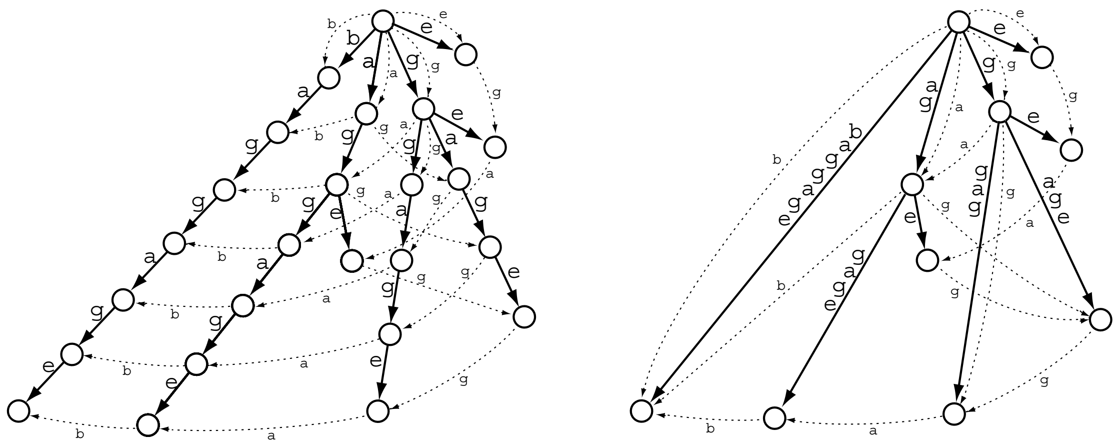


Figure 6.1: *STrie*(w) with *STrie*(w^{rev}) on the left, and *STree*(w) with *DAWG*(w^{rev}) on the right, where $w =$ `baggage`. The thick solid lines represent the edges of *STrie*(w) and *STree*(w), while the thin break lines do the ones of *STrie*(w^{rev}) and *DAWG*(w^{rev}). Since the string `baggage` ends with a unique character `e`, the end-marker \$ is omitted.

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

The edges $E_{R\to L}$ of Definition 6.2 are the so-called *shortest extension links* (*sext links*) of *STree*(w), which were introduced by Crochemore and Rytter in [14]. Moreover, a part of the reversed sext links are known as suffix links. Recalling the definition, the suffix links are the set

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

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

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

It can be observed that the suffix link set is a subset of the sext link set, under the '$\overset{w}{\overrightarrow{ax}} = a \cdot \overset{w}{\overrightarrow{x}}$'-condition.

In Fig. 6.1 we illustrate *STrie*(w) with *STrie*(w^{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}).

Now we pay our attention to *CDAWG*(w). Definition 2.10 can be transformed as follows:

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

In Definition 6.3, we show the definition of the symmetric CDAWG (SCDAWG) of a string w, denoted by $\mathit{SCDAWG}(w)$, originally defined by Blumer et al. [8].

Definition 6.3 *$\mathit{SCDAWG}(w)$ is the bidirectional dag $(V, E_{L\to R}, E_{R\to L})$ such that*

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

The edges $E_{R\to L}$ are called the sext links of $\mathit{CDAWG}(w)$, as well. The reversed suffix links of $\mathit{CDAWG}(w)$ are the set

$$\{(\overset{w}{\overleftrightarrow{x}}, \gamma a, \overset{w}{\overleftrightarrow{ax}}) \mid x, ax \in \mathit{Factor}(w), a \in \Sigma, \gamma \in \Sigma^*, \overset{w}{\overleftarrow{ax}} = \gamma ax, \overset{w}{\overleftrightarrow{x}} \neq \overset{w}{\overleftrightarrow{ax}}, \text{ and } \overset{w}{\overleftrightarrow{ax}} = a \cdot \overset{w}{\overleftrightarrow{x}}\}.$$

The suffix link set is a subset of the sext link set, under the '$\overset{w}{\overleftrightarrow{ax}} = a \cdot \overset{w}{\overleftrightarrow{x}}$'-condition.

We illustrate $\mathit{DAWG}(w)$ with $\mathit{STree}(w^{\mathrm{rev}})$, and $\mathit{SCDAWG}(w)$ in Fig. 6.2, where $w =$ `baggage`.

Another symmetric bidirectional index structure, called *affix tree*, was introduced by Stoye [50]. $\mathit{ATree}(w)$ and $\mathit{ATree}(w^{\mathrm{rev}})$ for $w =$ `baggage` are shown in Fig. 6.3 without a formal definition for comparison. Intuitively, the set of the nodes in $\mathit{SCDAWG}(w)$ is the *intersection* of those in $\mathit{STree}(w)$ and $\mathit{STree}(w^{\mathrm{rev}})$, while the set of the nodes in $\mathit{ATree}(w)$ is the *union* of them.

6.3 On-Line Construction of $\mathit{STree}(w)$ with $\mathit{DAWG}(w^{\mathrm{rev}})$

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

6.3.1 "$\mathit{STree}(w)$ with $\mathit{DAWG}(w^{\mathrm{rev}})$" Redefined

Our algorithm constructs $\mathit{STree}'(w)$ in the same fashion as the Ukkonen algorithm, and therefore the $\mathit{DAWG}(w^{\mathrm{rev}})$ being constructed at the same time is incomplete in the sense

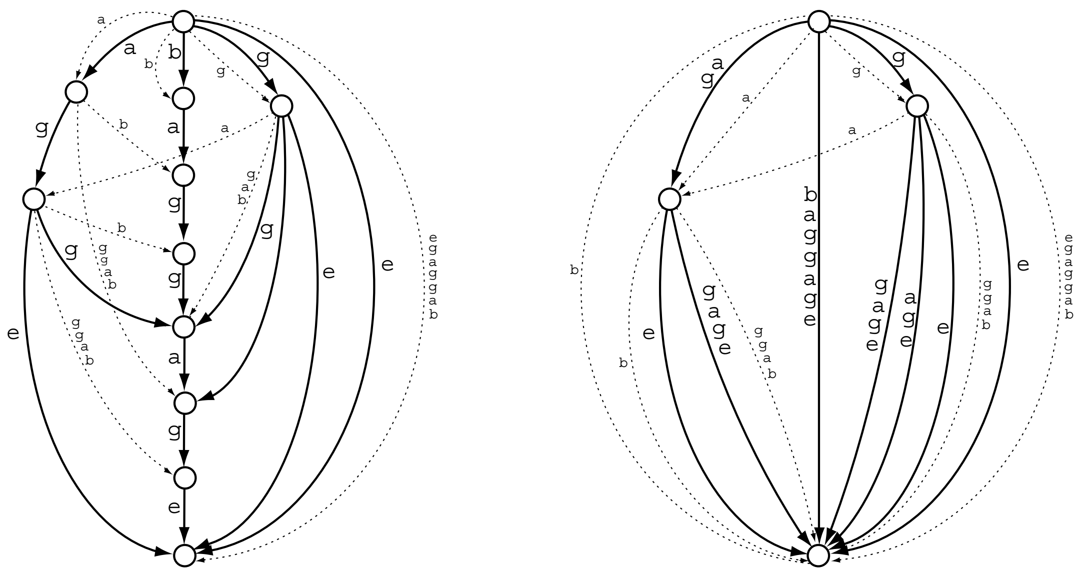


Figure 6.2: *DAWG*(w) with *STree*(w^{rev}) on the left, and *SCDAWG*(w) on the right, for string $w =$ `baggage`.

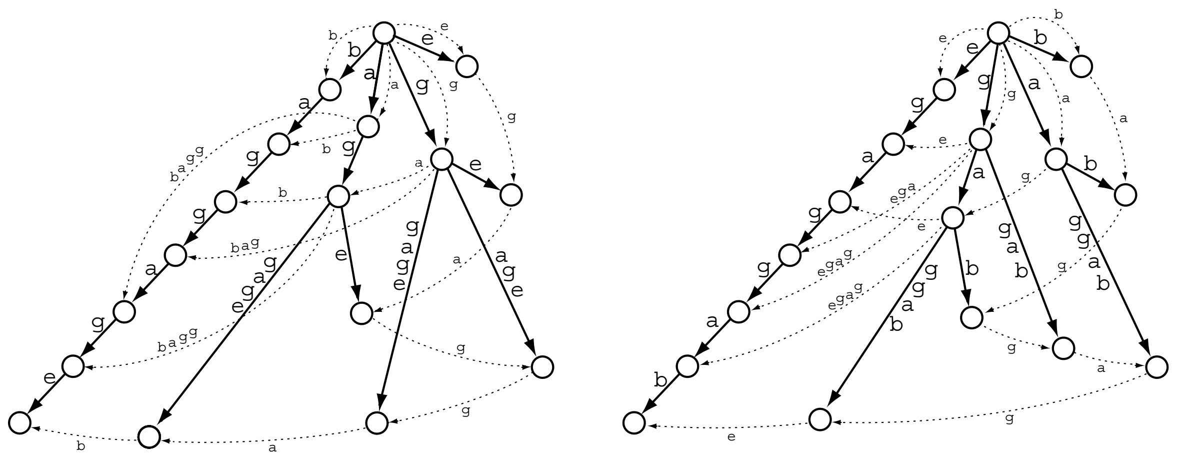


Figure 6.3: *ATree*(w) on the left and *ATree*(w^{rev}) on the right, where $w =$ `baggage`.

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*($\$w^{\mathrm{rev}}$).

Definition 6.4 *"STree(w) with sext links" is the bidirectional dag* $(V, E_{L\to R}, E_{R\to L})$ *such* *that*

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

Since $\overset{w\$}{\overrightarrow{x}} = \overset{w\$}{\overrightarrow{x}}$, this structure is identical to that defined in Definition 6.2 for input string $w\$$.

6.3.2 Main Idea of the Algorithm

As for $\mathit{STree}'(w)$ for a string $w \in \Sigma^*$, our algorithm creates it in entirely the same way as Ukkonen's algorithm (see Section 3.3 or [55]). Every time a new node is created during the construction of $\mathit{STree}'(w)$, the sext links of the new node, which correspond to certain edges of $\mathit{DAWG}(w^{\mathrm{rev}})$, are computed. Ukkonen's algorithm creates no leaf node for the use of so-called '∞-trick' that enables his algorithm to achieve an $O(|w|)$-time construction of $\mathit{STree}'(w)$, 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 in order 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 $\mathit{STree}'(w)$ can be quadratic due to a series of updating the open edges. However, recall the fact that label α of an edge of $\mathit{STree}'(w)$ 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 $\mathit{STree}'(w[1 : h])$ 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 an increment of the variable h.

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

Proposition 6.1 *Suppose that in $\mathit{STree}'(w)$ 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 $\mathit{STree}'(w)$.*

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

6.3.3 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. 6.4 that shows each phase of the construction of $STree'(\#\texttt{abab}\$)$. The starred point in Fig. 6.4 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 #`aba` and phase #`abab` in Fig. 6.4. 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. 6.5, left). Only v is the newly created node in case (b$_1$). Concrete examples can be seen in phases #, #`a`, #`ab`, and the third step of phase #`abab`\$ in Fig. 6.4. 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. 6.5, right). Concrete examples of case (b_2) can bee seen at the first and second steps in phase #`abab`\$ in Fig. 6.4. 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).

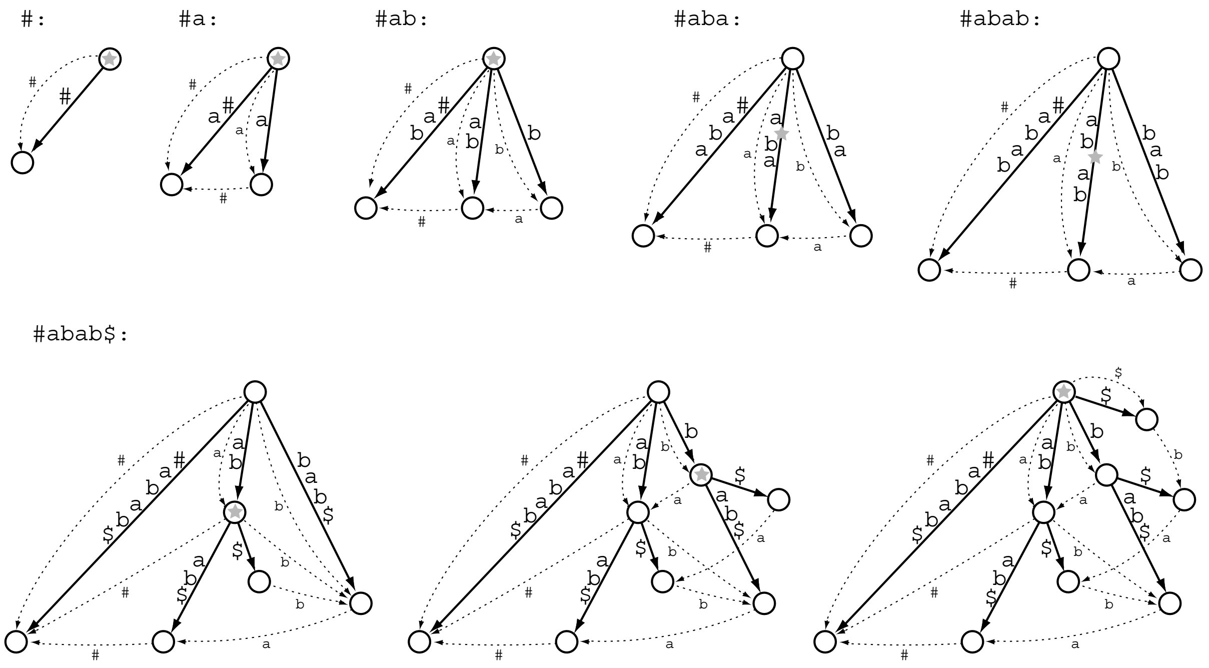


Figure 6.4: The on-line construction of $STree'$(#`abab`\$) with the sext links represented by the broken arrows. At the third step of phase #`abab`\$, the sext links form $DAWG$(\$`baba`#).

Sext Link of a Leaf Node

In both cases (b_1) and (b_2), it follows from Proposition 6.1 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

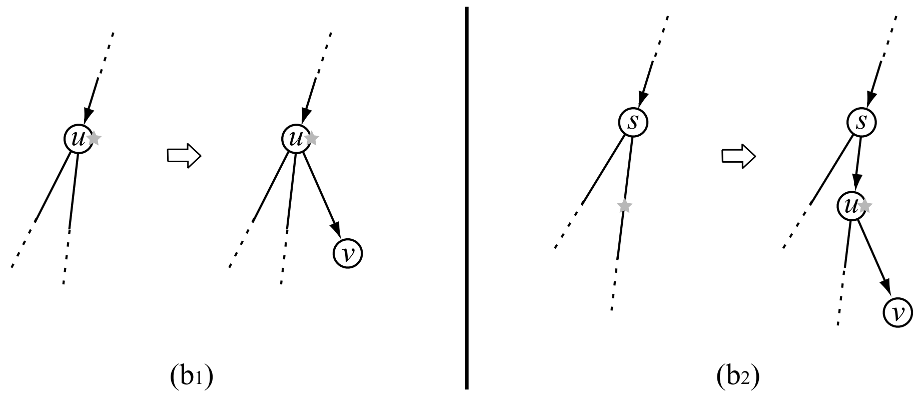


Figure 6.5: The two cases of the position of the active point, which is denoted by a gray star. Since the active point is on a node u in case (b_1) displayed on the left, only leaf node v is newly created. On the other hand, in case (b_2) on the right, internal node u is also created where the active point is at present, in the middle of an edge.

of v.

Proposition 6.2 *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. 6.4, 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\$.

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

$r' = \overset{w[1:i]}{\overrightarrow{bu}}$ in this case (remember the definition of sext links). For instance, at the first step of phase #abab\$ in Fig. 6.4, $u = \texttt{ab}$, $r = \texttt{abab\$}$ and $r' = \mathit{sext}[\texttt{abab\$}, \#] = \#\texttt{abab\$}$. Since $\mathit{rsuf}[\texttt{ab}, \#]$ is undefined, we define $\mathit{sext}[\texttt{ab}, \#] = \#\texttt{abab\$}$. If $b = a$, then $\mathit{sext}[u, b]$ stays pointing to node u', because obviously $u' = \overset{w[1:i]}{\overrightarrow{bu}} = \overset{w[1:i]}{\overrightarrow{au}}$. For example, at the second step of phase #abab\$ in Fig. 6.4, $\mathit{sext}[\texttt{bab\$}, \texttt{a}] = \texttt{abab\$}$. However, since $\mathit{rsuf}[\texttt{b}, \texttt{a}] = \texttt{ab}$, $\mathit{sext}[\texttt{b}, \texttt{a}] = \texttt{ab}$. As previously remarked in the reason (ii) in Section 6.3.2, 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 6.1 *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, $\mathit{sext}[u, c]$ is created for each character c such that either $\mathit{sext}[r, c]$ or $\mathit{sext}[v, c]$ 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 $\mathit{suf}[u] = u'$. Namely, node v currently represents $w[j : i+1]$ and node p corresponds to $w[j+1 : i+1]$. That is why $\mathit{sext}[p, w[j]] = v$. 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. 6.6.

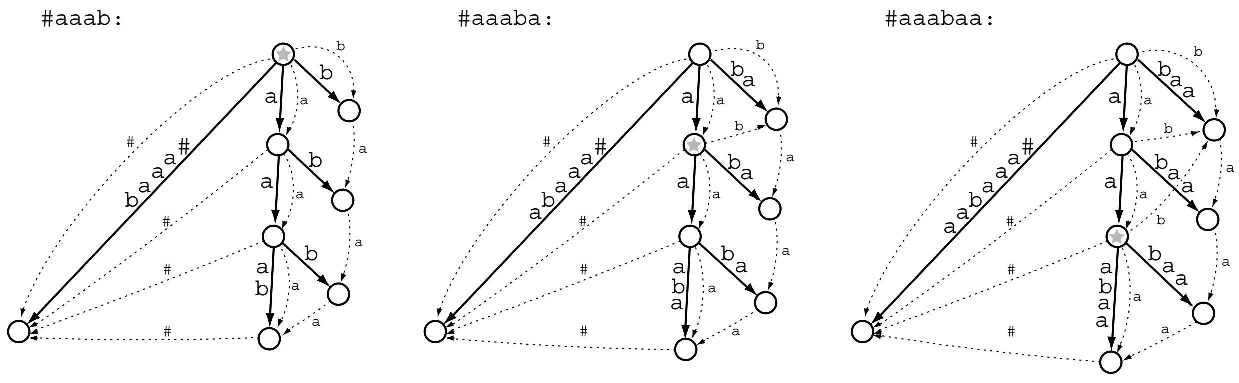


Figure 6.6: *STree*′(#aaab) with the sext links is shown on the left. Node b is the last created leaf node in that phase. Scanning a new character a, the active point moves to node a, as seen in the center figure *STree*′(#aaaba). Then, $sext$[a, b] is set to point to the last created leaf node ba. Also in the right figure representing *STree*′(#aaabaa), $sext$[aa, b] = baa, because the active point has arrived at node aa.

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. 6.4. 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 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, $\overset{w[1:j]}{\overrightarrow{cp}} = r$. If $|u| > |p|$, then the sext link of p is modified so as to point to u ($sext[p, c] = u$), because

$\overset{w[1:i]}{\overrightarrow{cp}} = u$. A concrete example can be seen between phase #`abab` and phase #`abab`\$ in Fig. 6.4. $sext[\varepsilon, \mathtt{a}] = \mathtt{abab}$ in phase #`abab` is modified as $sext[\varepsilon, \mathtt{a}] = \mathtt{ab}$ at the first step of phase #`abab`\$, where node `ab` is the internal node newly created in phase #`abab`\$. In another case (if $|u| \leq |p|$), the sext link of node p remains pointing to node r, since $\overset{w[1:i]}{\overrightarrow{cp}} = r$ in this case. Similar discussion holds for the sext links pointing to node v, another child of node u.

6.3.4 Correctness and Complexity of the Algorithm

The algorithm is summarized as Fig. 6.7. If we compute the sext links of the nodes in "$STree'(w)$ with sext links" according to the algorithm, we have the following:

Theorem 6.1 *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 [55] 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 $DAWG(w^{\mathrm{rev}})$ which corresponds to string w^{rev} is then completed. This guarantees that the paths of $DAWG(w^{\mathrm{rev}})$ corresponding to the suffixes of w^{rev} are also created as the sext links of the internal nodes of $STree'(w)$. This algorithm constructs $DAWG(w^{\mathrm{rev}})$ 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 algorithm 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. □

6.4 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 [32], and builds certain edges of $CDAWG(w^{\mathrm{rev}})$ as the sext links of the nodes of $CDAWG'(w)$.

We stress that the algorithm of [32] is based on the Ukkonen suffix tree construction algorithm. This implies, if we add the functions "*redirect*" and "*separate_node*" in [32] to the pseudo-code of the algorithm in Section 6.3, 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 [20] as an exercise. Furthermore, the CDAWG construction algorithm in [32] 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 6.3 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 $(r, (p, e), sink)$;" and labels of reversed suffix links and sext links can be of strings, not a character.

6.4.1 Sext Link Corresponding to a Newly Created Edge

A sequence of snapshots on the on-line construction of $SCDAWG'$(#abab$) is shown in Fig. 6.8. Since character a has appeared in string #a, the edge labeled with a is created and directed to the final node in phase #a. After that, the sext link of the initial node labeled with a# is set to point to the final node. Comparing it with the corresponding phase in Fig. 6.4, one can see that character # in the label a# of the CDAWG corresponds to the label # of the sext link from the leaf node a to node #a of the suffix tree in the phase #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 $(w[1:j])^{rev}$.

6.4.2 Change of Sext Links

See phases #abab and #abab$ in Fig. 6.8. The active point stays on the middle of the edge labeled abab in phase #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 a# is also cut into two. One labeled with a is set to point to the new node ab, and the other labeled # is set from node ab. It is because $\overset{w[1:6]}{\overrightarrow{\varepsilon a}} = ab$ and $\overset{w[1:6]}{\overrightarrow{\#ab}} = \#abab\$$ 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. 6.9 that displays $CDAWG'$(#abb) and $CDAWG'$(#abba). There is a sext link of the initial node pointing to the final node, which is labeled with ba# in $CDAWG'$(#abb). At the beginning of the conversion to $CDAWG'$(#abba), a new node b is created where the active point currently stays. Then, the sext link labeled ba# is cut and its former part is set to point to the new node b, labeled with 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 ba# with *string* bb in the final node, where bb is not the representative of the final node. This is because if we associate that just with the representative, like $\mathit{sext}[\varepsilon, \texttt{ba\#}] = \texttt{\#abb}$, 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 node). Therefore, we make a sext link point to a string represented in a certain node, not to the representative. For example, on $\mathit{CDAWG}'(\texttt{\#abb})$ in Fig. 6.9, $\mathit{sext}[\varepsilon, \#] = \texttt{\#abb}$, $\mathit{sext}[\varepsilon, \texttt{a\#}] = \texttt{abb}$, $\mathit{sext}[\varepsilon, \texttt{ba\#}] = \texttt{bb}$.

As seen in phase #abab$ of Fig. 6.8, the edge labeled with bab$ is *merged* into the node ab and its label is modified to b. According to it, $\mathit{sext}[\varepsilon, \texttt{ba\#}] = \texttt{bab\$}$ becomes $\mathit{sext}[\varepsilon, \texttt{ba}] = \texttt{b}$. 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 $\mathit{STree}(\texttt{\#abab\$})$ in Fig. 6.4.

Fig. 6.10 displays a node *separation* that can happen during the construction of CDAWGs. In Fig. 6.10, 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 $\mathit{CDAWG}'(\texttt{\#ababcb})$. Then, $\mathit{sext}[\varepsilon, \texttt{ba}] = \texttt{b}$ in $\mathit{CDAWG}'(\texttt{\#ababc})$ is cut into two, one of which is $\mathit{sext}[\texttt{b}, \texttt{a}] = \texttt{ab}$ and the other $\mathit{sext}[\varepsilon, \texttt{b}] = \texttt{b}$.

6.4.3 Implementation of Factors Represented in a Node

As is mentioned above, a sext link in $\mathit{CDAWG}'(w)$ 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 $\mathit{CDAWG}'(w[1:i])$ 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. 6.10, $\mathit{CDAWG}'(\texttt{\#ababc})$. 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 $\mathit{CDAWG}'(\texttt{\#ababc})$, 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 6.2 *For any string $w \in \Sigma^*$, SCDAWG for w can be constructed on-line in linear time and space with respect to $|w|$.*

6.5 Conclusion

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 [55], and simultaneously builds the DAWG as the sext links of the nodes of the suffix tree.

Blumer et al. [8] 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 space at the same time when constructing an SCDAWG.

```
Algorithm Construction of $STree'(w\$)$ with sext links
in alphabet $\Sigma = \{w[-1], \ldots, w[-m]\}$.
 1 create nodes root and $\perp$;
 2 for $j := 1$ to $m$ do create a new edge $(\perp, (-j, -j), root)$;
 3 $suf[root] := \perp$;
 4 $length(root) := 0$;    $length(\perp) := -1$;
 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 $w[i] = \$$;

function $update(s, (k, p), lastleaf, n)$:
 1 $oldr :=$ nil; $s' :=$ nil;
 2 while not $check\_end\_point(s, (k, p-1), w[p])$ do
 3     if $k \leq p - 1$ then    /* implicit */
 4         $s' := extension(s, (k, p-1))$;
 5         $r := split\_edge(s, (k, p-1))$;
 6     else $r := s$;    /* explicit */
 7     create a new leaf node $v$ and a new edge $(r, (p, e), v)$;
 8     /* e is the global variable representing the scanned length of the input string. */
 9     let $length(v)$ be $e - n$;
10     if $oldr \neq$ nil then $set\_suffix\_link(oldr, r)$;
11     if $lastleaf \neq$ nil then $set\_suffix\_link(lastleaf, v)$;
12     if $r \neq s$ then    /* maintenance of sext links */
13         $c := w[n]$;
14         if $rsuf[r, c] =$ nil then $sext[r, c] := sext[v, c]$;
15         for each character $a$ such that $sext[s', a] \neq$ nil do
16             if $rsuf[r, a] =$ nil then $sext[r, a] := sext[s', a]$;
17         for each sext link $sext[x, a] = s'$ do /* modify sext links pointing to s' */
18             if $length(r) > length(x)$ then $sext[x, a] := r$;
19     $oldr := r$;    $lastleaf := v$;    $n := n + 1$;
20     $(s, k) := canonize(suf[s], (k, p-1))$;
21 if $oldr \neq$ nil then $set\_suffix\_link(oldr, s)$;
22 $(s, k) := canonize(s, (k, p))$;
23 if $k > p$ then $sext[s, w[n]] := lastleaf$;
24 return $(s, k, lastleaf, n)$;

procedure $set\_suffix\_link(s, t)$:
 1 let $c$ be the first character of the string represented by $s$;
 2 $suf[s] := t$;    $rsuf[t, c] := s$;    $sext[t, c] := s$;
```

Figure 6.7: The algorithm to construct "*STree*$'(w)$ with sext links". *check_end_point*, *extension*, *canonize*, and *split_edge* are identical to those used in Chapter 3.

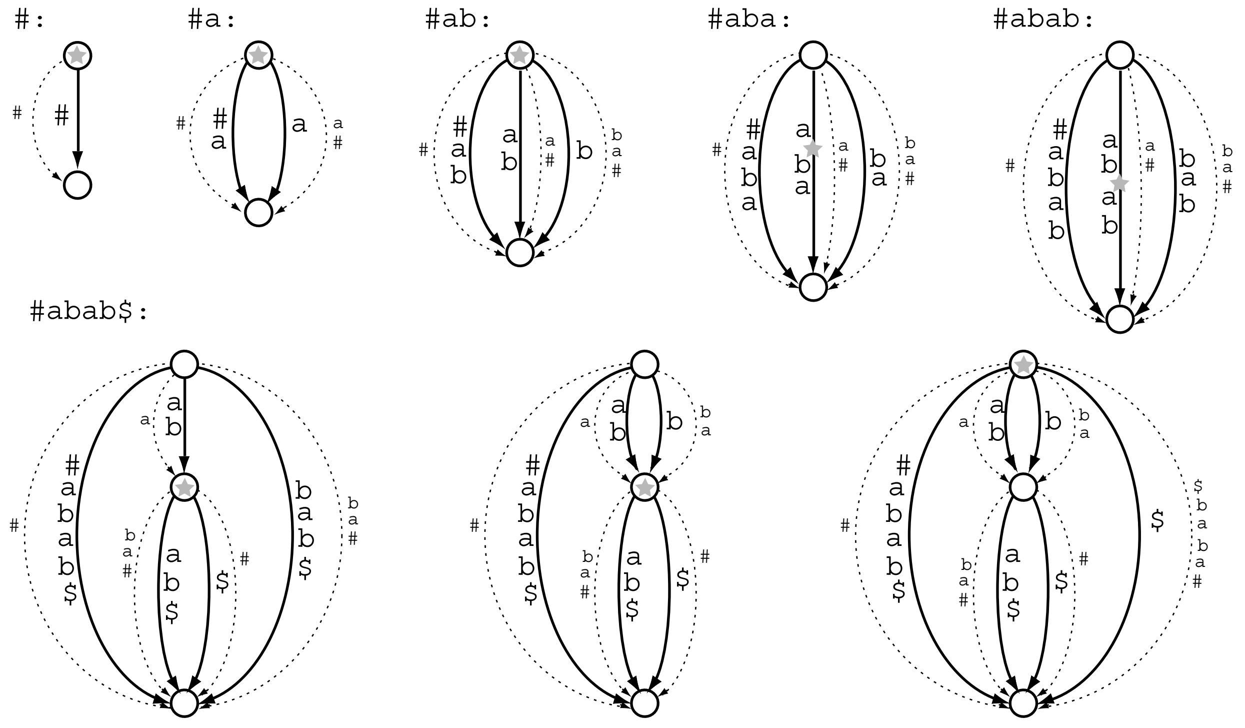


Figure 6.8: The on-line construction of $SCDAWG'(w)$, where $w = \#\mathtt{abab}\$$. The solid arrows represent the edges of $CDAWG'(w)$, whereas the broken arrows represent the sext links of the nodes of $CDAWG'(w)$, that are equivalent to the edges of $CDAWG(w^{rev})$.

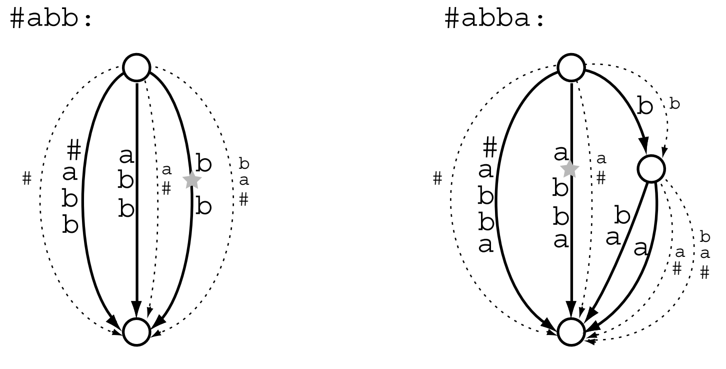
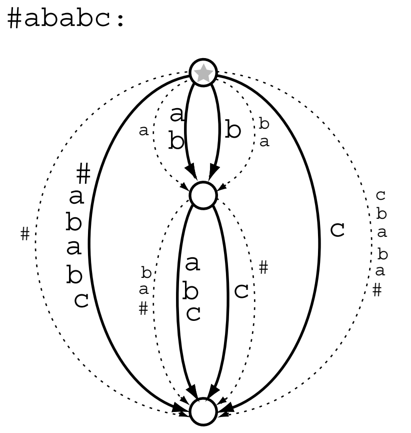


Figure 6.9: $CDAWG'(\#\mathtt{abb})$ and $CDAWG'(\#\mathtt{abba})$ with their sext links. The sext link $sext[\varepsilon, \mathtt{ba}\#]$ is cut into two sext links $sext[\varepsilon, \mathtt{b}]$ and $sext[b, \mathtt{a}\#]$.



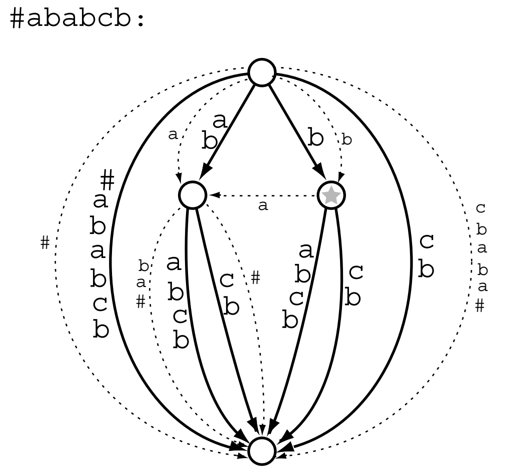


Figure 6.10: *CDAWG*′(#ababc) and *CDAWG*′(#ababcb) with their sext links.

Chapter 7

Bidirectional Construction of Index Structures

7.1 Background and Motivation

As repeatedly remarked in the previous sections, the invention of Ukkonen's algorithm has allowed us to update an input string by adding new strings afterward. Without his algorithm, we would have to construct the suffix tree from scratch. However, since his algorithm is supposed to only read a string from left to right, even it is not capable of permitting us to update the input string to the left direction. Namely, if wanting to extend the current input string by appending another string to its front, we still now have to reconstruct the suffix tree from the beginning. As typically seen in Bioinformatics, a considerable amount of strings, like DNA sequences for example, has to be treated. It is easy to imagine that reconstructing the suffix tree of such a quite long string is a very huge task.

In this chapter we give the very one to settle the matter above. Namely, that is an algorithm to construct a suffix tree *bidirectionally*, both left to right and right to left. The idea of updating a suffix tree from right to left is based on Weiner's algorithm. However, his original algorithm is not suitable for updating a suffix tree which is specialized to be treaded by Ukkonen's algorithm. Thereby we modify Weiner's algorithm. Mutually, Ukkonen's algorithm is also modified according to the influence from Weiner's algorithm.

As once mentioned in Chapter 6, Chen and Seiferas first declared the duality of suffix trees and DAWGs [11]. For any string w, the suffix tree of w and the DAWG of w^{rev} can share the same nodes, where w^{rev} is the reversed string of w. In [30] the on-line algorithm to construct both the suffix tree for w and the DAWG for w^{rev} was given, whose basis is Ukkonen's suffix tree construction algorithm. In this sense the DAWG of string $u = w^{rev}$ can be constructed on-line, from right to left. Also, it is remarked in [14] that Weiner's algorithm can be modified so to build the DAWG of w^{rev} together with the suffix tree of w. Therefore it can build the DAWG of string $u = w^{rev}$ from left to right. It results in that the algorithm we present in this paper can also build the DAWG of a given string bidirectionally, on-line.

Some related work can be seen in literature: Stoye [50] invented a variant of suffix trees, called *affix trees*. He gave an algorithm for bidirectional construction of affix trees, and Maaß improved the time complexity of the algorithm to $O(n)$ [39].

The result of this chapter was published in [28].

7.2 Bidirectional Construction of Suffix Trees

7.2.1 Left Extension

The part to extend a given suffix tree with a character added at the left of the string is based on Weiner's algorithm [58]. This operation is denoted by *Left_Extension*.

Weiner's Algorithm.

For a smart implementation of the algorithm, we use the *bottom node* Ukkonen also used in his algorithm [55]. For any string $w \in \Sigma^*$, we suppose that between the bottom node and the root node of $STree'(w)$ there is an edge labeled by symbol Σ, which means any string in the alphabet. The main idea of Weiner's algorithm is as follows. Suppose that we are given string $w \in \Sigma^*$ for which we are constructing the suffix tree. Weiner's algorithm reads the string from right to left. The algorithm constructs $STree'(w[i-1:n])$ from $STree'(w[i:n])$ for $n+1 \geq i \geq 2$, by inserting prefixes of $w[i-1:n]$ into $STree'(w[i:n])$. We divide the prefixes of $w[i-1:n]$ into the following two groups.

(1) Prefixes $w[i-1:h]$ for $0 \leq h \leq j$ where j is the largest number satisfying $w[i-1:j] \in Factor(w[i:n])$.

(2) Prefixes $w[i-1:h]$ for $j+1 \leq h \leq n$.

Any prefixes in the group **(1)** do not need to be newly inserted into $\mathit{STree}'(w[i:n])$, simply because they are already represented in the suffix tree. We call the longest string in the group **(1)**, $w[i-1:j+1]$, the *head* for $w[i-1:n]$, and denote it by $\mathit{head}_{w[i-1:n]}$. The prefixes in the group **(2)** are inserted into the suffix tree from the location associated with $\mathit{head}_{w[i-1:n]}$. Since at least the empty string ε belongs to the group **(1)**, there always exists the location for $\mathit{head}_{w[i-1:n]}$ in $\mathit{STree}'(w[i:n])$. We start the search of $\mathit{head}_{w[i-1:n]}$ from the leaf node corresponding to the longest suffix of $w[i:n]$, which is $w[i:n]$ itself. We call it the *active leaf* and denote it by $\mathit{active\_leaf}_{w[i-1:n]}$. The path from it to the bottom node is called the *working path*. The search starts from $\mathit{active\_leaf}_{w[i-1:n]}$ and proceeds along the working path up to the bottom node. Naive method to localize it takes quadratic time. To avoid it, we use two dimensional tables *test* and *link*, both of whose first components are characters and second are nodes.

Definition 7.1 *Let w be an arbitrary string in Σ^* and a be an arbitrary character in Σ. Assuming that v is a node in $\mathit{STree}'(w)$ (i.e. $\overset{\xrightarrow{w}}{v} = v$), then:*

1. *$\mathit{test}[a,v] = \textbf{true}$ iff $av \in \mathit{Factor}(w)$.*

2. *$\mathit{link}[a,v] = u$ if $u = av$ for some node u in $\mathit{STree}'(w)$.*
   Otherwise, $\mathit{link}[a,v] = \textbf{nil}$.

Notice that $\mathit{test}[a,v] = \textbf{true}$ if $\mathit{link}[a,v] \neq \textbf{nil}$ for any character $a \in \Sigma$ and node v in $\mathit{STree}'(w)$. It is easy to see that *link* equals the reversed (and labeled) link of a suffix link of $\mathit{STree}'(w)$ for any string $w \in \Sigma^*$(see Definition 2.11). Let the *first_test* be the node on the working path and nearest to $\mathit{active\_leaf}_{w[i-1:n]}$ with which the table *test* for $w[i-1]$ is **true**, and denote it by $\mathit{first\_test}_{w[i-1:n]}$. Similarly, let the *first_link* be the node on the working path and nearest to $\mathit{active\_leaf}_{w[i-1:n]}$ with which the table *link* for $w[i-1]$ is u for some node u, and denote it by $\mathit{first\_link}_{w[i-1:n]}$. Let α be the concatenation of labels of edges between $\mathit{first\_test}_{w[i-1:n]}$ and $\mathit{active\_leaf}_{w[i-1:n]}$, and β the one from $\mathit{first\_link}_{w[i-1:n]}$ to $\mathit{first\_test}_{w[i-1:n]}$. Note that α can never be ε whereas β might be ε. Remark that $\mathit{head}_{w[i-1:n]} = \mathit{link}[w[i-1], \mathit{first\_link}_{w[i-1:n]}]$. When $\mathit{first\_test}_{w[i-1:n]} \neq \mathit{first\_link}_{w[i-1:n]}$ ($\beta \neq \varepsilon$) and $\mathit{head}_{w[i-1:n]}$ is on an edge (r, γ, s), the edge is split into the two edges $(r, \beta, \mathit{head}_{w[i-1:n]})$ and $(\mathit{head}_{w[i-1:n]}, \delta, s)$, where $\gamma = \beta\delta$. These are followed by the creation of the new edge $(\mathit{head}_{w[i-1:n]}, \alpha, new\_leaf_{w[i-1:n]})$, where $new\_leaf_{w[i-1:n]}$ denotes the leaf node newly created. Because $new\_leaf_{w[i-1:n]}$ corresponds to $w[i-1:n]$ and all suffixes of $w[i-1:n]$ other than $w[i-1:n]$ itself have already been inserted, it is the completion of the construction of $\mathit{STree}'(w[i-1:n])$. Then, the tables *test* and *link* are maintained. Obviously, $\mathit{new\_leaf}_{w[i-1:n]}$ becomes $\mathit{active\_leaf}_{w[i-2:n]}$ in the next step $i-2$.

Modifying Weiner's Algorithm.

Consider the longest factor y of $w[i:n]$ such that $y \in \mathit{Suffix}(w[i-1:n])$. We call it the *active point* of $\mathit{STree}'(w[i-1:n])$ the location y corresponds to, and denote it by $\mathit{active\_point}_{w[i-1:n]}$. Notice that $\mathit{active\_point}_{w[i-1:n]}$ is the longest suffix of $w[i-1:n]$ not represented by a leaf node.

Lemma 7.1 *Let $w \in \Sigma^*$. Suppose that we are updating $\mathit{STree}'(w[i:n])$ to $\mathit{STree}'(w[i-1:n])$, where $n=|w|$ and $n+1 \geq i \geq 2$. Let $\mathit{head}_{w[i-1:n]} = ax$ with $a \in \Sigma$ and $x \in \Sigma^*$. If $ax \in \mathit{Suffix}(w[i-1:n])$, then x is always $\mathit{active\_point}_{w[i-1:n]}$.*

Let $xy = \mathit{active\_leaf}_{w[i-1:n]}$ for some string $y \in \Sigma^+$. Consider the case that $\overset{\overset{w}{\Longrightarrow}}{x} \neq x$. (Then there is no node corresponding to $\mathit{active\_point}_{w[i:n]}$ in $\mathit{STree}'(w[i:n])$.) It implies that $\mathit{first\_test}_{w[i-1:n]}$ is the root node and $\mathit{first\_link}_{w[i-1:n]}$ is the bottom node. If $x \neq \varepsilon$, then $\mathit{head}_{w[i-1:n]} = ax$ cannot be localized by the information from $\mathit{first\_test}_{w[i-1:n]}$ and $\mathit{first\_link}_{w[i-1:n]}$. To avoid this trouble, we treat the active point as though there exists a node for $x = \mathit{active\_point}_{w[i:n]}$. Concretely, the values of tables *test* and *link* are computed also for $\mathit{active\_point}_{w[i:n]}$.

When $\mathit{active\_point}_{w[i-1:n]} = x$ is on an edge (r, α, s) in $\mathit{STree}'(w[i-1:n])$, it is represented by a *reference pair* (r', δ) such that r' is an ancestor of r and $r'\delta = x$. For the string β such that $r\beta = x$, (r, β) is called the *canonical* reference pair of $\mathit{active\_point}_{w[i-1:n]}$.

When $\mathit{STree}'(w[i:n])$ is updated to $\mathit{STree}'(w[i-1:n])$, the active point is maintained as follows. Let $\mathit{active\_point}_{w[i:n]} = u$. The suffix of $w[i:n]$ corresponding to $\mathit{active\_point}_{w[i:n]}$ is $w[\ell:n]$, where $\ell = n - |u| + 1$. There are three cases to consider:

(Case 1) $w[\ell:n]$ is not a prefix of $w[i:n]$.

(Case 2) $w[\ell:n]$ is a prefix of $w[i:n]$ but $w[\ell-1:n]$ is not a prefix of $w[i-1:n]$.

(Case 3) $w[\ell:n]$ is a prefix of $w[i:n]$ and $w[\ell-1:n]$ is a prefix of $w[i-1:n]$.

In both **(Case 1)** and **(Case 2)**, $\mathit{active\_point}_{w[i-1:n]} = \mathit{active\_point}_{w[i:n]}$. Meanwhile, in **(Case 3)**, $\mathit{active\_point}_{w[i-1:n]}$ corresponds to au where $a = w[i-1]$.

To implement the suffix tree using only linear space, an edge (r, α, s) in $STree'(w[i:n])$ is actually implemented as $(r, (k, p), s)$ such that $w[k:p] = \alpha$, where $k \geq i$ and $p \leq n$. In case that s is a leaf node, $p = n$. However, we cannot specify p since the suffix tree might be also extended to the right direction by *Right_Extension* to be introduced in the next section. If specifying the value of p, it becomes impossible that the algorithm runs in linear time. Therefore, we adopt Ukkonen's so-called "∞-trick" [55]. That is to say, we implement the edge as $(r, (k, \infty), s)$ such that $w[k:n] = \alpha$.

Theorem 7.1 *For any string $w \in \Sigma^*$, the modified Weiner's algorithm constructs $STree'(w)$ on-line (right to left) and in linear time with respect to $|w|$.*

The construction of $STree'$(cocoo) by *Left_Extension* is displayed in Fig 7.1.

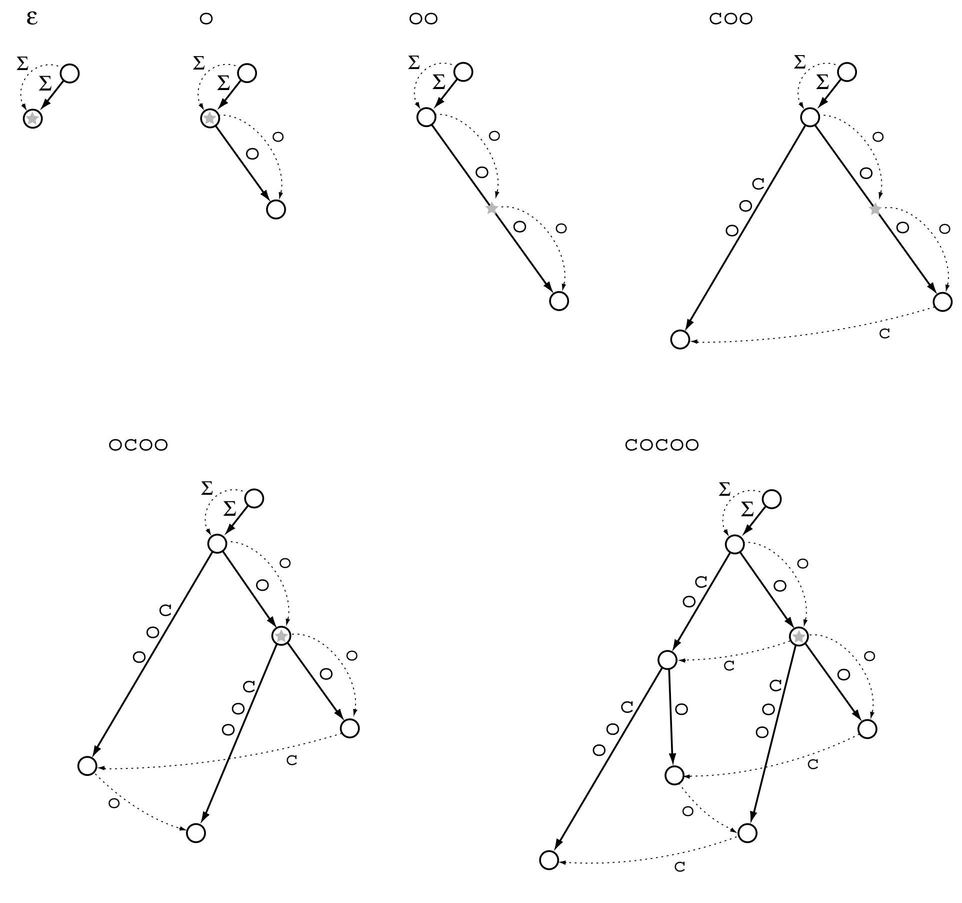


Figure 7.1: Construction of $STree'$(cocoo) by *Left_Extension*. The gray star denotes the active point for each step. The broken arrows represent the links for the table *link*.

7.2.2 Right Extension

The part to extend a given suffix tree with a new character added to the right of the string is based on Ukkonen's algorithm [55]. This operation is denoted by *Right_Extension*.

See Section 3.3 where Ukkonen's algorithm has been recalled.

Modifying Ukkonen's Algorithm.

We have to modify Ukkonen's algorithm so that it also computes the tables *test* and *link*, which are used by *Left_Extension*.

Lemma 7.2 *Ukkonen's algorithm can be modified so as to compute the tables test and link during the construction of* $STree'(w)$ *for any* $w \in \Sigma^*$. *The overall time complexity is still* $O(|w|)$.

We here omit the detail of proving the above lemma, but remark that computing the tables *test* and *link* is inherently the same thing as computing the *sext links*, abbreviation for the *shortest extension links*, introduced in [14]. It was showed in [30] that a modified Ukkonen's algorithm can compute the sext links in $O(|w|)$ time. As for the active leaf stated in the previous section, it is obvious that $active\_leaf_{w[1:i]} = active\_leaf_{w[1:i+1]}$ for any $1 \leq i \leq n-1$. Therefore we need no maintenance for it.

The construction of $STree'(\texttt{cocoo})$ by *Right_Extension* is displayed in Fig 7.2.

7.2.3 Bidirectional On-Line Construction of Suffix Trees

We here explain how our algorithm constructs a suffix tree in *bidirectional* on-line manner. Given a string $w \in \Sigma^*$, the algorithm can get the construction started with any position i in w, where $1 \leq i \leq |w|$. Firstly $STree'(w[i:i])$ is constructed, by either *Left_Extension* or *Right_Extension*. Then by *Left_Extension* $STree'(w[i:i])$ is updated with characters $w[j]$ where j is from $i-1$ down to 1, and by *Right_Extension* with characters $w[k]$ where k is from $i+1$ up to n. For any j and k the structure built by the algorithm is $STree'(w[j:k])$. Here a symbol \$ appearing nowhere in w is appended after w if $w[n]$ is not a unique character in w. Since $STree'(w\$) = STree(w\$)$, after the construction we obtain inherently the same structure as $STree(w)$ for any string w. In Fig. 7.3 an example of bidirectional construction of $STree'(\texttt{cocoon})$ is shown.

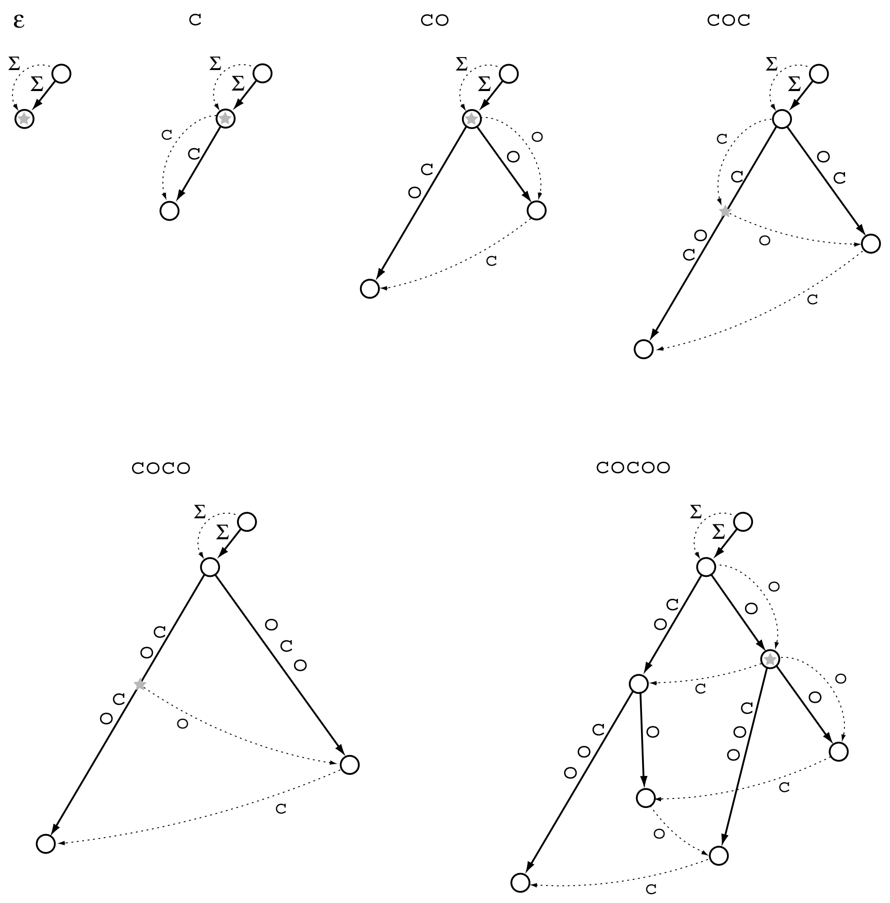


Figure 7.2: Construction of *STree*′(cocoo) by *Right_Extension*. The gray star denotes the active point for each step. Here only links for the table *link* are drawn as broken arrows, so the links of the table *suf* correspond to the broken arrows in the reversed direction.

Theorem 7.2 *For any string $w \in \Sigma^*$ the proposed algorithm constructs STree′(w) in bidirectional on-line manner and in time linear in $|w|$.*

Proof. We briefly prove that from any position of a given string the suffix tree can be built in linear time.

Given a string $w \in \Sigma^*$, suppose that we get the construction started with the position i in w, where $1 \leq i \leq |w|$. Let $|w| = n$. Suppose we here construct *STree*′$(w[i : i])$ by *Right_Extension*. Then the suffix tree is updated by *Right_Extension* $(n - i + 1)$ times. The matter needed to be considered is the cost to compute the active point. Though the active point is computed $(n - i + 1)$ times, it might take more than $O(n - i + 1)$ time if the suffix tree is extended also by *Left_Extension* during the update by *Right_Extension*.

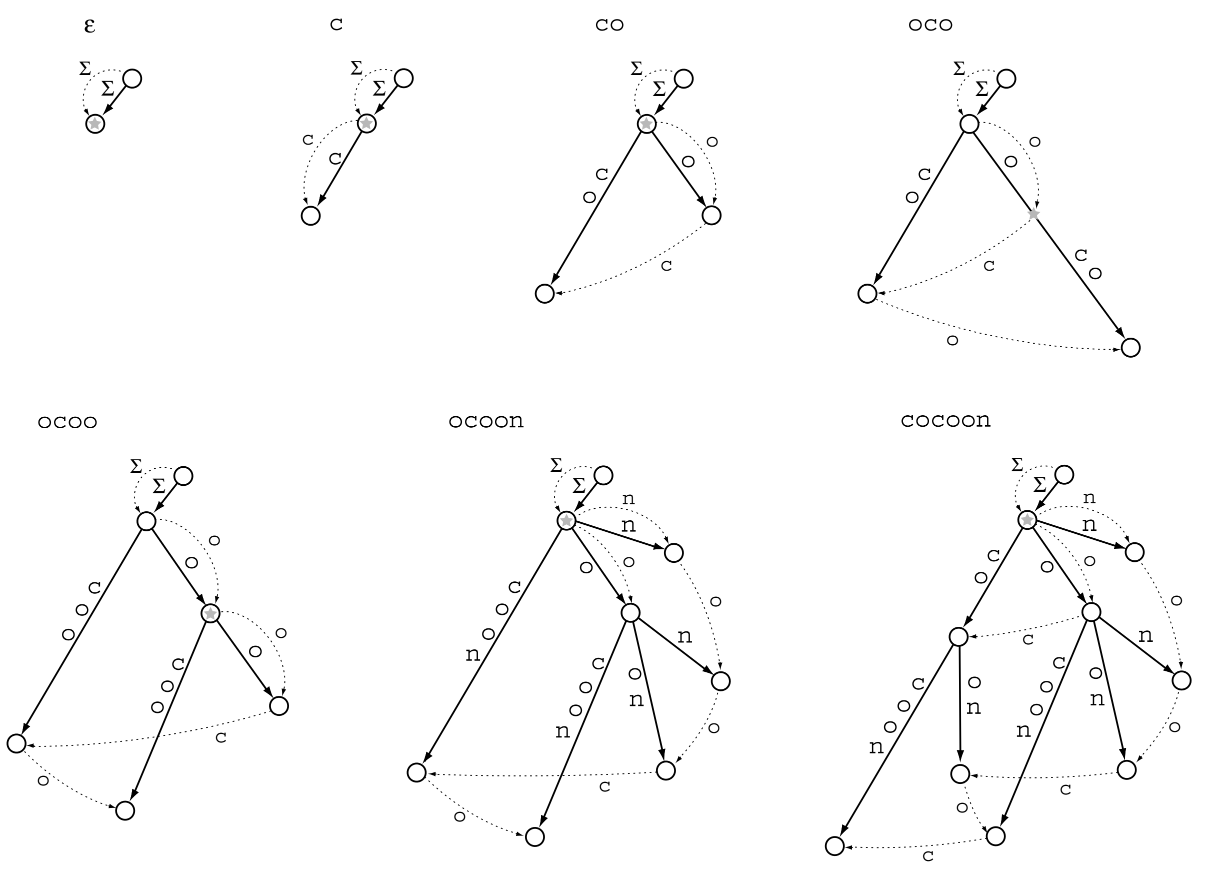


Figure 7.3: Bidirectional Construction of *STree*′(`cocoon`). The construction has begun with `c` in the position 3 of the input string `cocoon`.

However, the cost is bounded by $O(n-i+1+k)$, where k $(0 \leq k \leq i-1)$ is the number of update of the suffix tree by *Left_Extension*. Consequently, the cost of *Right_Extension* is $O(n)$ for any i with $1 \leq i \leq n$.

From now on we consider the cost of *Left_Extension*. Suppose that we now have *STree*′$(w[k:j])$ with $1 \leq k \leq i-1$ and $i \leq j \leq n$, that is, the suffix tree has been updated $(i-k)$ times by *Left_Extension* and $(j-i+1)$ times by *Right_Extension*. Then, the cost of the next iteration by *Left_Extension* can be charged to the difference between the depths of $active\_leaf_{w[k:j]}$ and $active\_leaf_{w[k-1:j]}$ (for detail see [14]). Thus the sum of all these differences is proportional to j. Since $j \leq n$, the cost of *Left_Extension* is $O(n)$ for any i with $1 \leq i \leq n$. □

7.3 Bidirectional Construction of DAWGs

The directed acyclic word graph (DAWG) is a well known and widely studied index structure, originally introduced in [7]. The DAWG is the smallest finite state automaton recognizing all suffixes of a given string [12].

Let us go back to the suffix trees. We introduce a table called the *shortest extension links*, the *sext links* for short, of $STree'(w)$ for a string $w \in \Sigma^*$. The idea of the sext link was originally given in [14], but it was for $STree(w)$. What we treat here is one for $STree'(w)$.

Definition 7.2 *Let w be an arbitrary string in Σ^* and a be an arbitrary character in Σ. For a string $x \in Factor(w)$, $sext[a, \overset{w}{\overrightarrow{\overrightarrow{x}}}] = \overset{w}{\overrightarrow{\overrightarrow{ax}}}$.*

It derives from Definition 7.2 that the sext links can be described as the set $T = \{(\overset{w}{\overrightarrow{\overrightarrow{x}}}, a, \overset{w}{\overrightarrow{\overrightarrow{ax}}}) \mid x, ax \in Factor(w) \text{ and } a \in \Sigma\}$. Assume that a string w ends with a unique character. On this assumption $\overset{w}{\overrightarrow{\overrightarrow{x}}} = \overset{w}{\overrightarrow{x}}$ for any string $x \in Factor(w)$. Therefore, the following set T' is identical to T for any such string w, $T' = \{(\overset{w}{\overrightarrow{x}}, a, \overset{w}{\overrightarrow{ax}}) \mid x, ax \in Factor(w) \text{ and } a \in \Sigma\}$. By considering the reversal w^{rev} of the string w, the set T' can be transformed as $T' = \{(\overset{z}{\overleftarrow{y}}, a, \overset{z}{\overleftarrow{ya}}) \mid y, ya \in Factor(z), a \in \Sigma, \text{ and } z = w^{rev}\}$. Observe that there is a trivial one-to-one correspondence between E in Definition 2.9 and T'. Hence the following lemma stands.

Lemma 7.3 *Let w be an arbitrary string in Σ^* whose right most character is unique in w. The set T of sext links of w is identical to the set E of edges of $DAWG(w)$.*

Meanwhile, the table *sext* has a very close correspondence with tables *test* and *link*. Let v be a node of $STree'(w)$ and a be a character in Σ, then;

(1) if $sext[a, v] = \textbf{nil}$, then $test[a, v] = \textbf{false}$.

(2) else $test[a, v] = \textbf{true}$. Let $sext[a, v] = u$, then

  (a) if $|u| = |v| + 1$, then $link[a, v] = u$.

  (b) else $link[a, v] = \textbf{nil}$.

If we use table *length* for a node x whose value is $|x|$, it is possible to examine the condition of **(2)-(a)**, whether $|u| = |v| + 1$, in constant time. It implies that when constructing $STree'(w)$ in bidirectional on-line manner, we can use the table *sext* instead of the tables *test* and *link*. When we are given a string $w \in \Sigma^*$, we consider the string $z = w^{\mathrm{rev}}$ and use a symbol \$ occurring nowhere in z, as $z\$$. It then follows from Theorem 7.2 and Lemma 7.3 that:

Theorem 7.3 *For any string $w \in \Sigma^*$, the algorithm presented in Section 7.2 can construct $DAWG(w)$ in bidirectional on-line manner and in time linear in $|w|$.*

It is shown in Fig 7.4 how $DAWG(w)$ is constructed to both direction, where $w =$ cocoo.

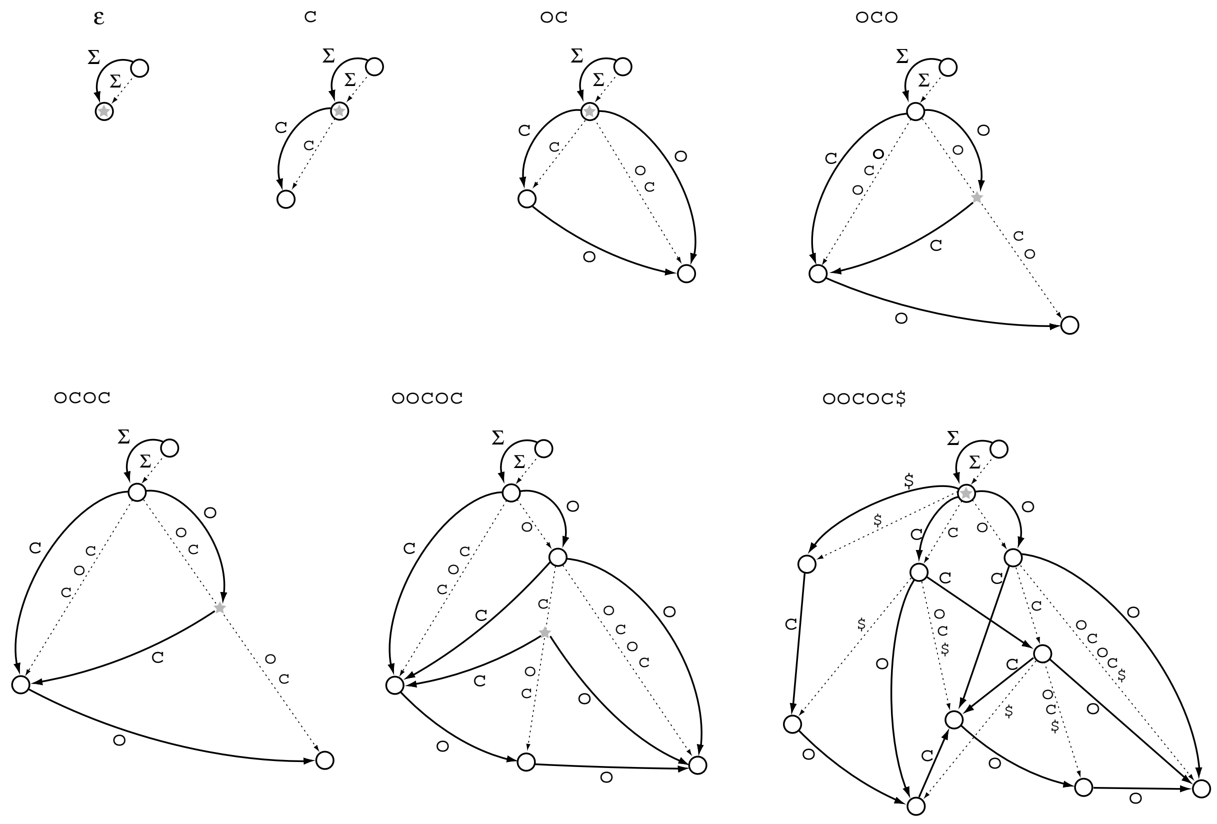


Figure 7.4: We are given string cocoo and now constructing $DAWG$(cocoo). In so doing, we build $STree'$(oococ\$) while computing the sext links. The broken edges represent the edges of the suffix tree, whereas the solid edges do the sext links. Ignoring the path \$co from the root node, the sext links in $STree'$(oococ\$) are equal to the edges of $DAWG$(cocoo).

7.4 Concluding Remarks

We developed a bidirectional, on-line, and linear-time suffix tree construction algorithm. We also showed that the algorithm can also construct DAWGs in bidirectional on-line manner and in linear time with respect to the length of an input string.

It might be interesting to consider bidirectional construction of compact directed acyclic word graphs (CDAWGs), which require smaller space than both suffix trees and DAWGs [8, 16]. Extending a given CDAWG to the right direction should be based on the on-line (left to right) algorithm introduced in [32]. The left extension of a CDAWG could perhaps be accomplished by a modified version of Weiner's algorithm, which is going to be our future work.

Chapter 8

The Minimum All-Suffixes Directed Acyclic Word Graphs

8.1 Introduction

A variety of patterns have been considered so far, according to various kinds of purposes and aims. The most basic one is a *substring* pattern. Let Σ be a finite alphabet. We call an element in Σ a *character*, and one in Σ^* a *string*. We say a pattern string p is a substring of a text string w if $w = upv$ for some strings $u, v \in \Sigma^*$. When a text w is fixed and a pattern p is flexible, once constructing a suitable data structure for w, we can solve the substring matching problem in $O(|p|)$ time, where $|p|$ denotes the length of p. In order to solve the problem efficiently, much attention has extensively been paid to inventing efficient data structures, such as suffix trees [58, 43, 55], directed acyclic word graphs (DAWGs) [7, 12], compact directed acyclic word graphs (CDAWGs) [8, 16, 32], suffix arrays [41], compact suffix arrays [40], suffix cacti [35], compressed suffix arrays [47, 19], that are also mentioned in previous chapters.

Meanwhile, the problem finding a *subsequence* pattern has also been widely studied. We say a pattern p is a subsequence of a text w if p can be obtained by removing zero or more characters from w. By means of the directed acyclic subsequence graph (DASG) for w, we can examine whether or not p is a subsequence of w in $O(|p|)$ time [4, 15]. An *episode* pattern is a "length-bounded" version of a subsequence pattern [42], as mentioned in Chapter 9. An episode pattern is given in the form of a pair of a string p and an integer k, as $\langle p, k \rangle$. If p is a subsequence of x such that x is a substring of w with $|x| \leq k$, we say that the episode pattern $\langle p, k\rangle$ matches w. The episode directed acyclic subsequence graphs (EDASGs) were given in [53], for a practical solution of the problem.

Now we propose a new kind of pattern matching problem: *Given a text string* $w = w_1w_2\cdots w_n$*, a string* p *and an integer* i*, examine whether or not* p *is a substring of* $w[i:]$ *where* $w[i:] = w_i \dots w_n$ *(NOTE: if* $i > |w|$*, the answer is always NO).* We name the pattern $\langle p, i\rangle$ a *beginning-sensitive pattern*, a *BS-pattern* for short. For any string $w \in \Sigma^*$ we denote by $DAWG(w)$ the DAWG of w. Using the DAWGs for all suffixes of w, this problem is solvable in $O(|p|)$ time. This simple collection of the DAWGs is called the *naive all-suffixes directed acyclic word graph* for w, written as the naive $ASDAWG(w)$. Since the size of $DAWG(w)$ is $O(|w|)$, that of the naive $ASDAWG(w)$ is $O(|w|^2)$.

In this paper we introduce a new data structure, named the *minimum* $ASDAWG(w)$ and denoted by $MASDAWG(w)$. $MASDAWG(w)$ is the minimization of the naive $ASDAWG(w)$. We show that the size of $MASDAWG(w)$ is $\Theta(|w|)$ if $|\Sigma| = 1$, and $\Theta(|w|^2)$ if $|\Sigma| \geq 2$. Also, we produce an *on-line* algorithm that *directly* constructs $MASDAWG(w)$ in time linear in the size of $MASDAWG(w)$.

We show further two applications of $MASDAWG(w)$, one of which is called the *region sensitive* pattern matching problem: *Given a text string* $w = w_1w_2\cdots w_n$*, a string* p *and integers* i, j*, examine whether or not* p *is a substring of* $w[i:j]$ *where* $w[i:j] = w_i \dots w_j$ *(NOTE: if* $i > |w|$*, the answer is always NO).* We name the pattern $\langle p, (i, j)\rangle$ a *region-sensitive pattern*, an *RS-pattern* for short. Using $MASDAWG(w)$ with additional data, the RS-pattern problem is solvable in $O(|p|)$ time.

Finding a good rule to separate given two sets of strings, often referred to as *positive examples* and *negative examples*, is a critical task in Knowledge Discovery and Data Mining. In [22], an efficient method, with which a subsequence pattern is considered as a rule for the separation, was given, and in [23] one using an episode pattern was proposed (see also Chapter 9). $MASDAWG(w)$ is believed certainly to be a good "weapon" to develop a practical algorithm to find the best VLDC-patterns to distinguish given two sets of strings efficiently. In fact, our experimental result has shown that the average size of the minimum ASDAWGs for random texts of length 1 to 500 over a binary alphabet is proportional to $|w|^{1.24}$, in spite of the theoretical space complexity, $\Theta(|w|^2)$.

The result was published in [33].

8.2 All-Suffixes Directed Acyclic Word Graphs

In this chapter we introduce a new structure named the *all-suffixes directed acyclic word graph* (*ASDAWG* for short). Also, the minimized version of the structure, called the *minimum ASDAWG* (*MASDAWG* for short), is also defined. We give a tight bound for the space complexity of the MASDAWG for an input string.

Throughout this chapter, let $w[i:] = w[i:|w|]$ for $1 \leq i \leq |w|+1$.

Definition 8.1 (All-Suffixes DAWG (ASDAWG)) *$ASDAWG(w)$ is a kind of deterministic automaton with $|w|+1$ initial states, designated by integers $0, 1, \ldots, |w|$, in which the subgraph consisting of the states reachable from an initial state k and of their outgoing edges is $DAWG(w[k+1:])$.*

The simple collection of $DAWG(w[1:])$, $DAWG(w[2:])$,…, $DAWG(w[n])$, $DAWG(w[n+1:])$ $(n = |w|)$ is an example of $ASDAWG(w)$, to which we refer as the *naive* $ASDAWG(w)$. The number of nodes of the naive $ASDAWG(w)$ is $\Theta(|w|^2)$. By minimizing the naive $ASDAWG(w)$, we can obtain the *minimum* $ASDAWG(w)$, which is denoted by $MASDAWG(w)$. The naive $ASDAWG(abba)$ and $MASDAWG(abba)$ are shown in Fig. 8.1. The minimization is performed based on the equivalence relation defined as follows. Let denote a node $[x]^R_u$ of $DAWG(u)$ by an ordered pair $\langle u, [x]^R_u \rangle$. Every node of the naive $ASDAWG(w)$ can be represented by a pair $\langle u, [x]^R_u \rangle$ with $u \in \mathit{Suffix}(w)$ and $x \in \mathit{Factor}(u)$. The equivalence relation, denoted by $\sim_w$, is defined by

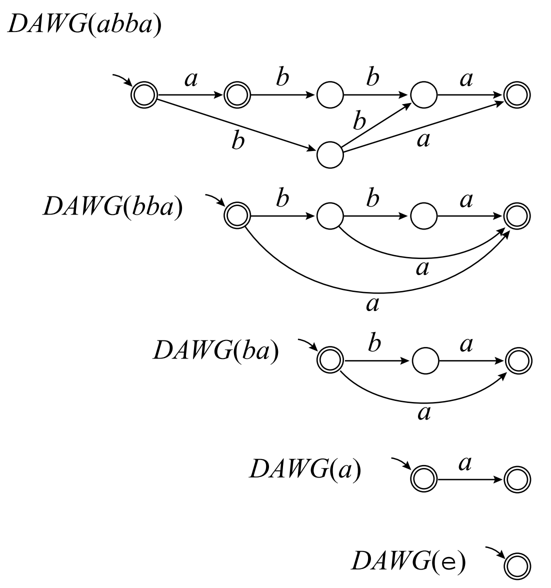
$$\langle u, [x]^R_u \rangle \sim_w \langle v, [y]^R_v \rangle \Leftrightarrow x^{-1}\mathit{Suffix}(u) = y^{-1}\mathit{Suffix}(v)$$

A node of $MASDAWG(w)$ corresponds to an equivalence class under $\sim_w$. We write $\langle u, [x]^R_u \rangle$ simply as $\langle u, [x] \rangle$ if no confusion occurs.

Proposition 8.1 *Let $u \in \mathit{Suffix}(w)$. Let x be a nonempty factor of u. We factorize u as $u = hxt$ and assume h is the shortest such string. Then, $\langle hxt, [x] \rangle$ is equivalent to $\langle sxt, [x] \rangle$ for every suffix s of h. (NOTE: The string x is not necessarily the representative of $[x]^R_u$.)*

Let $h_0, h_1, \ldots, h_r$ be the suffixes of the string h arranged in the decreasing order of their length. The above proposition implies an existence of the chain of equivalent nodes

$$\langle h_0xt, [x] \rangle, \langle h_1xt, [x] \rangle, \ldots, \langle h_rxt, [x] \rangle.$$



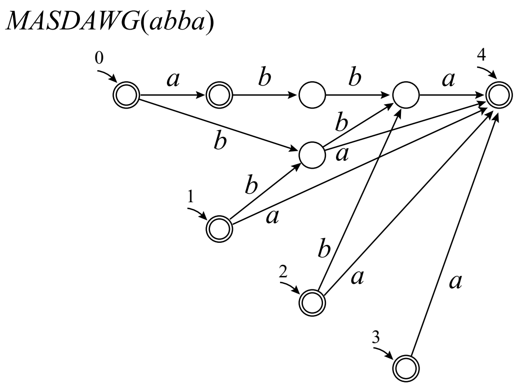


Figure 8.1: On the left $DAWG(x)$ for any string x in $Suffix(abba)$ are shown, and the collection of those is the naive $ASDAWG(w)$. On the right $MASDAWG(w)$ is displayed. While there are 16 states and 16 transitions in the former in total, there are 9 states and 12 transitions in the latter. For example, the nodes $\langle abba, [b]\rangle$ and $\langle bba, [b]\rangle$ are equivalent due to Case 1 and merged into one. Also, $\langle abba, [abb]\rangle$, $\langle bba, [bb]\rangle$, and $\langle ba, [b]\rangle$ are merged into one node, where the first two are equivalent due to Case 2 and the last two are equivalent due to Case 3. The upper four sink nodes are equivalent due to Case 2 and the lowest one is equivalent to them (see Lemma 8.2), and therefore the five are merged into one sink node.

In case more than one string exist in $[x]^R_u$, the chain length r is maximized if we choose the shortest one as x. The chain, however, does not necessarily break at the node $\langle h_r xt, [x]\rangle$. The shortest string in $[x]^R_u$ is not necessarily the shortest in $[x]^R_{h_r xt}$: Shorter one may exist. Thus we need more precise discussion.

Lemma 8.1 *Let $h \in \Sigma^+$ and $u, hu \in Suffix(w)$. If a node of $DAWG(u)$ is equivalent to some node of $DAWG(hu)$, then it is also equivalent to some node of $DAWG(au)$ where a is the last character of the string h.*

Proof. Let $h = ta$ ($t \in \Sigma^*$). Assume $t \neq \varepsilon$. Let $x \in Factor(u)$ with $x \neq \varepsilon$, and $y \in Factor(tau)$ with $y \neq \varepsilon$. Assume $x^{-1}Suffix(u) = y^{-1}Suffix(tau)$. We have two cases to consider.

- $x \equiv_u^R y$. In this case, every occurrence of the string y within tau must be included within the u part. Thus, we have $x^{-1}\mathit{Suffix}(u) = y^{-1}\mathit{Suffix}(au)$.

- $x \not\equiv_u^R y$. In this case, (1) y is written as $y = sx$ where s is a nonempty string, and (2) there is an occurrence of y within tau that covers the boundary between a and u but the x part of the occurrence of $y = sx$ is contained in the u part of the string tau. In this case, by truncating an appropriate length prefix of s we can obtain a string z as a suffix of $y = sx$ such that $x^{-1}\mathit{Suffix}(u) = z^{-1}\mathit{Suffix}(au)$.

The proof is now complete. □

The above lemma guarantees that the DAWGs sharing one node of *MASDAWG*(w) are 'consecutive.' We therefore concentrate on the relation between two consecutive DAWGs. First, we consider the equivalence of the initial state.

Lemma 8.2 *Suppose $b \in \Sigma$ and $u, bu \in \mathit{Suffix}(w)$. Let $y \in \mathit{Factor}(bu)$ and assume y is the representative of $[y]_{bu}^R$. Then, the node $\langle u, [\varepsilon]\rangle$ and $\langle bu, [y]\rangle$ are equivalent under $\sim_w$ if and only if $y = b$ and u is of the form b^ℓ with $\ell \geq 0$.*

See, for example, *MASDAWG*$(bbbbb)$ shown in Fig. 8.2.

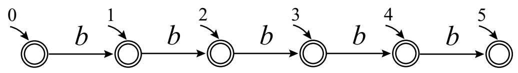


Figure 8.2: *MASDAWG*(w) for $w = b^5$. For every $i = 0, 1, \ldots, 4$, the initial node $[\varepsilon]_{b^i}^R$ of *DAWG*(b^i) is equivalent to the node $[b]_{b^{i+1}}^R$ of *DAWG*(b^{i+1}).

As an extreme case of Lemma 8.2 where $\ell = 0$, the node $[\varepsilon]_\varepsilon^R$ of *DAWG*(ε) is always equivalent to the sink node $[b]_b^R$ of the previous *DAWG*(b).

Next, we consider the case of the states but the initial state.

Lemma 8.3 *Suppose $b \in \Sigma$ and $u, bu \in \mathit{Suffix}(w)$. Let $x \in \mathit{Factor}(u)$ with $x \neq \varepsilon$. Let $y \in \mathit{Factor}(bu)$ with $y \neq \varepsilon$. Assume x and y are the representatives of $[x]_u^R$ and $[y]_{bu}^R$, respectively. The equivalence $\langle u, [x]\rangle \sim_w \langle bu, [y]\rangle$ implies that if $y \in \mathit{Prefix}(bu)$ then $y = bx$ and $x \in \mathit{Prefix}(u)$, and otherwise $y = x$. Moreover, $\langle u, [x]\rangle \sim_w \langle bu, [y]\rangle$ holds if and only if either*

(Case 1) *$x \notin$ Prefix(bu) and $y = x$;*

(Case 2) *$x \in$ Prefix(u), $x \equiv^R_{bu} y$, and $y = bx$; or*

(Case 3) *$x = b^i$, $y = b^{i+1}$, and u is of the form $b^\ell s$ such that $i \le \ell$, and $s \in \Sigma^*$ does not begin with b and does not contain an occurrence of b^i.*

Proof. Suppose $x^{-1}\mathit{Suffix}(u) = y^{-1}\mathit{Suffix}(bu)$. Let $u[i+1:]$ $(0 < i \le |u|)$ be the longest member of this set.

1. When $y \in \mathit{Prefix}(bu)$. Then, $i = |y| - 1$ and $y = by'$ with $y' = u[1:i]$. Since $u[i+1:] \in \mathit{Suffix}(x)$, we have $u = hxu[i+1:]$ for some $h \in \Sigma^*$. Namely, x is a suffix of $y' = u[1:i]$.

   (a) When $y' \notin \mathit{Prefix}(bu)$. We have $y \equiv^R_{bu} y'$ and

   $$(y')^{-1}\mathit{Suffix}(u) = (y')^{-1}\mathit{Suffix}(bu) = y^{-1}\mathit{Suffix}(bu) = x^{-1}\mathit{Suffix}(u),$$

   which implies $x \equiv^R_u y'$. Since $y' \in \mathit{Prefix}(u)$, y' must be the representative of $[y']^R_u = [x]^R_u$, thus we have $x = y'$.

   (b) When $y' \in \mathit{Prefix}(bu)$. String y' is a prefix of $y = by'$, and therefore has a period of 1. Hence we have $y' = b^i$ and $y = b^{i+1}$. Since x is a suffix of $y' = b^i$, $x = b^j$ for some j with $0 < j \le i$. If $j < i$, then $u[j+1:] \in x^{-1}\mathit{Suffix}(u)$, a contradiction. Thus we have $j = i$, i.e., $x = b^i$. On the other hand, $u[1:i] = y' = b^i$ and thus u is of the form $b^\ell s$ such that $\ell \ge i$ and $s \in \Sigma^*$ does not begin with b. We can show that the string s cannot contain an occurrence of $x = b^i$.

   Note that we have $x \in \mathit{Prefix}(u)$ in both the cases.

2. When $y \notin \mathit{Prefix}(bu)$. We have $y^{-1}\mathit{Suffix}(u) = y^{-1}\mathit{Suffix}(bu) = x^{-1}\mathit{Suffix}(u)$, which implies $x \equiv^R_u y$. From the choice of x, y must be a suffix of x and $x = \delta y$ with $\delta \in \Sigma^*$. Assume, for a contradiction, that $x^{-1}\mathit{Suffix}(bu) \ne x^{-1}\mathit{Suffix}(u)$. Then there must be a suffix $u[j+1:]$ of u such that $j < i$ and $bu = hxu[j+1:]$ with $h \in \Sigma^*$. Since $x = \delta y$, we have $bu = h\delta y u[j+1:]$, which implies $u[j+1:] \in y^{-1}\mathit{Suffix}(bu)$, a contradiction. Hence we have $x \equiv^R_{bu} y$. From the choice of y, x must be a suffix of y. Thus we have $x = y$.

It should be noted that Case 1 and Case 2 of Lemma 8.3 fit to Proposition 8.1, whereas Case 3 is irregular in the sense that the two equivalence classes $[x]^R_u$ and $[y]^R_{bu}$ have no common member despite $\langle u, [x]\rangle \sim_w \langle bu, [y]\rangle$. See Fig. 8.1, which includes instances of Case 1, Case 2, and Case 3.

The *owner* of a node of $\mathit{MASDAWG}(w)$ is defined to be the $\mathit{DAWG}(w[k:])$ such that k is the smallest integer for which $\mathit{DAWG}(w[k:])$ shares the node. We are now ready to estimate the lower bound of the number of nodes of $\mathit{MASDAWG}(w)$.

Theorem 8.1 *When $|\Sigma| \geq 2$, the number of nodes of $\mathit{MASDAWG}(w)$ for a string w is $\Theta(|w|^2)$. It is $\Theta(|w|)$ for a unary alphabet.*

Proof. The proof for the case of a unary alphabet $\Sigma = \{a\}$ is not difficult. We can use Lemma 8.2. We now prove the lower bound in the case of $|\Sigma| \geq 2$. Let us consider a string $w = (ab)^m(ba)^m$, where a, b are distinct characters from Σ. For each $i = 2, \ldots, m-1$, let $u_i = (ab)^i(ba)^m$. Let $x = (ba)^j$ with $0 < j < i$. It is not difficult to show that $x \not\equiv^R_{u_i} ax$ and $x \not\equiv^R_{u_i} b^{-1}x$, and therefore $[x]^R_{u_i} = \{x\}$. Thus x is the representative of $[x]^R_{u_i}$, and we can use the above lemma. Since $x \in \mathit{Prefix}(bu_i)$, $x \notin \mathit{Prefix}(u_i)$, and the first character of u_i is not b, none of the three conditions is satisfied, and therefore $\mathit{DAWG}(u_i)$ is the owner of the node corresponding to $[x]^R_{u_i}$. Thus, the nodes of $\mathit{MASDAWG}(w)$ corresponding to

$$[(ba)^1]^R_{u_i}, [(ba)^2]^R_{u_i}, \ldots, [(ba)^{i-1}]^R_{u_i}$$

are distinct and are owned by $\mathit{DAWG}(u_i)$. For each i with $1 < i < m$, $\mathit{DAWG}(u_i)$ has at least $i-1$ own nodes. Thus, $\mathit{MASDAWG}(w)$ has $\Omega(m^2) = \Omega(|w|^2)$ nodes. □

8.3 On-Line Construction of MASDAWGs

Since the construction of the naive $\mathit{ASDAWG}(w)$ takes $O(|w|^2)$ and the minimization can be performed in linear time proportional to the number of the edges of the naive $\mathit{ASDAWG}(w)$ (see [46]), we can build $\mathit{MASDAWG}(w)$ in $O(|w|^2)$. On the other hand, we have shown that the number of nodes in $\mathit{MASDAWG}(w)$ is $\Theta(|w|^2)$. We are therefore interested in on-line and direct construction of $\mathit{MASDAWG}(w)$. We have obtained the following result.

Theorem 8.2 *$\mathit{MASDAWG}(w)$ can be constructed directly and on-line in time linear with respect to its size.*

The algorithm for on-line construction of $MASDAWG(w)$ basically simulates the on-line constructions of the DAWGs for all suffixes of a string w. Figure 8.3 illustrates the on-line construction of $MASDAWG(abbab)$.

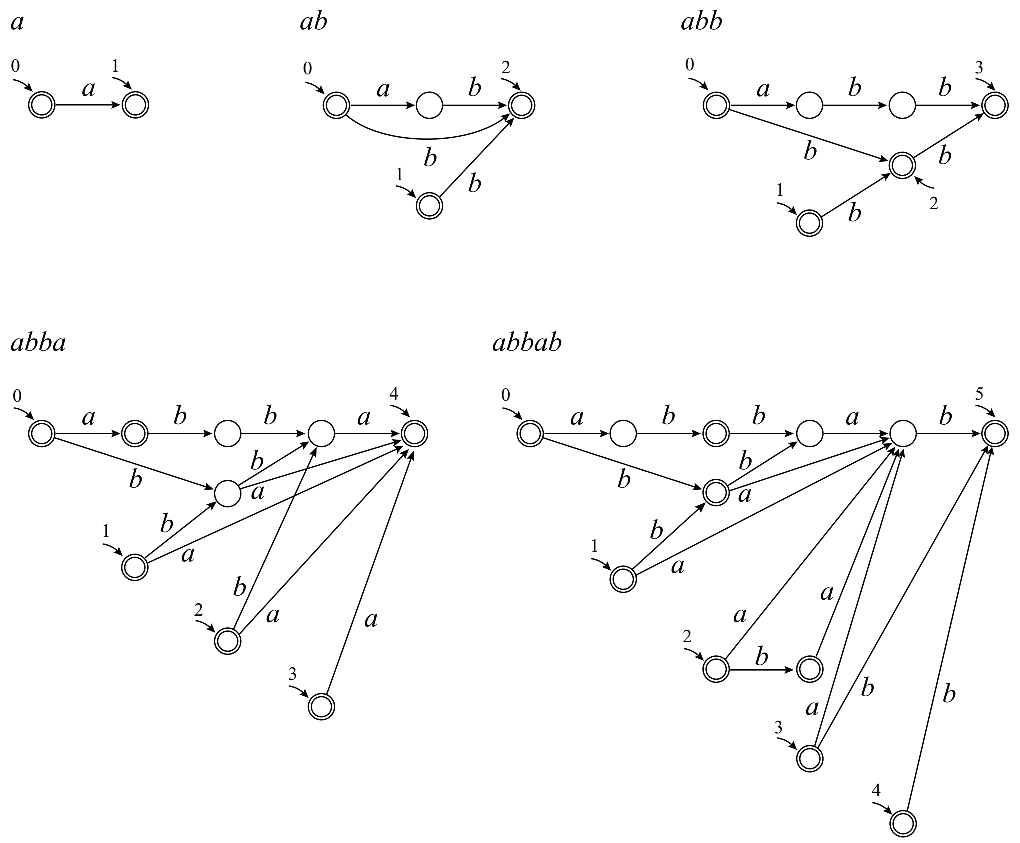


Figure 8.3: On-line construction of $MASDAWG(w)$ for $w = abbab$. Each initial state becomes independent whenever the newly appended character violates the condition of Lemma 8.2. Node separation of other type occurs only twice. One happens during the update of $MASDAWG(ab)$ to $MASDAWG(abb)$. The sink node consisting of $\langle abb, [ab]\rangle$ and $\langle b, [b]\rangle$ is separated into two nodes. This is recognized as a node separation in $DAWG(abb)$. The other occurs during the update of $MASDAWG(abba)$ to $MASDAWG(abbab)$. The node consisting of $\langle abba, [abb]\rangle$, $\langle bba, [bb]\rangle$, and $\langle ba, [b]\rangle$ is separated into two. This is a special case in the sense that no node separation occurs inside any of $DAWG(abba)$, $DAWG(bba)$, and $DAWG(ba)$. (See the first case of Lemma 8.7.) (Note: Though each accepting state is marked double-circled in any step in this figure, we do not maintain it on-line. After the construction of $MASDAWG(w)$ is completed, we mark every node reached during the suffix-links-traversal from the sink node.)

In the following sections, we present a basic idea of the algorithm together with several lemmas which support it.

8.3.1 Suffix Links

In the construction, the *suffix links* play a key role. One main difference compared with constructing a single DAWG is that a node may have more than one suffix link. This happens because $MASDAWG(w)$ may contain two distinct, equivalent nodes $\langle u, [x] \rangle$ and $\langle v, [y] \rangle$ such that the node to which the suffix link from $\langle u, [x] \rangle$ points is not equivalent to the node to which the suffix link from $\langle v, [y] \rangle$ points. We update $MASDAWG(w)$ into $MASDAWG(wa)$ as if the underlying DAWGs for $w[1:], w[2:], \ldots$ were updated simultaneously, as follows. Conceptually, we reserve all suffix links of these DAWGs, by associating each suffix link with the corresponding DAWG. Whenever two or more suffix links are duplicated, the corresponding DAWGs are consecutive due to Lemma 8.1, so that we can handle them at once. This is critical for the linearity of our algorithm. We traverse the dag induced by the suffix links rooted from the sink node, in the order of the corresponding DAWGs, and process each encountered node appropriately (creating a new edge to the new sink node, separating the node, or redirecting an edge to the separated node).

8.3.2 Compact Representation of Node Length Information

Remember in the on-line construction of the DAWG for a single string, there occurs an event so-called *node separation*. Formally, this event is described as follows. We store in each node $[x]^R_w$ of $DAWG(w)$ its *length*, namely, the length of the representative of $[x]^R_w$. Consider updating $DAWG(w)$ to $DAWG(wa)$ where a is a character. Let z be the longest suffix of wa that also occurs within w. We call it the *longest repeated suffix* of wa. A node separation happens iff z is not the representative of $[z]^R_w$. The node $[z]^R_w$ can be detected by traversing the suffix link chain from the sink in order to find its parent node $[z']^R_w$, which is the first encountered node on the chain that has an outgoing edge labelled by a. Whenever the length of $[z]^R_w$ is greater than that of its parent $[z']^R_w$ plus one, the node $[z]^R_w$ of $DAWG(w)$ is separated into two nodes $[x]^R_{wa}$ and $[z]^R_{wa}$ of $DAWG(wa)$, where x is the representative of $[z]^R_w$.

Note that a node of $MASDAWG(w)$ corresponds to an equivalence class under the equivalence relation $\sim_w$, and therefore two or more DAWGs may share a node of $MASDAWG(w)$. We need to know the length of the corresponding node of an arbitrary one among them. Naive solution would be to store into a node of $MASDAWG(w)$ a $(|w|+1)$-tuple of integers, the ith value of which indicates the length of the corresponding node of the i-th DAWG, where $i = 0, 1, \ldots, |w|$. The space requirement is, however, proportional to $|w|^3$. Below we give an idea of compact representation of the tuple.

Lemma 8.4 *Let $\langle w[i+1:], [x_1]\rangle, \ldots, \langle w[i+\ell:], [x_\ell]\rangle$ be the nodes of the naive $ASDAWG(w)$ which are merged into one node in $MASDAWG(w)$, where $0 \le i$ and $i+\ell \le |w|+1$. We assume each of the strings $x_1, \ldots, x_\ell$ is the representatives of the equivalence class of it. Then, there exists an integer k with $1 \le k \le \ell$ such that*

$$x_j = \begin{cases} x_k, & \text{if } 1 \le j \le k; \\ x_k[j-k+1:], & \text{if } k < j \le \ell. \end{cases}$$

(See Fig. 8.4.)

Proof. By Lemma 8.3. □

For example, $MASDAWG(abb)$ in Fig. 8.3 has a node consisting of $\langle abb, [b]\rangle$ and $\langle bb, [b]\rangle$. Also, $MASDAWG(abba)$ has a node consisting of $\langle abba, [abb]\rangle$, $\langle bba, [bb]\rangle$, and $\langle ba, [b]\rangle$.

It follows from the above lemma that the function that takes as input an integer s and returns $|x_s|$ if $1 \le s \le \ell$ can be represented as a quartet $(i, \ell, k, |x_k|)$, which requires only a constant space (or $O(\log |w|)$ space). The update procedure of the quartet for each node is basically apparent, except for the nodes in which node separations occur.

8.3.3 Node Separation

Recall that two or more DAWGs can share one node of $MASDAWG(w)$, and each of them has a possibility of being separated into two nodes. This seems to complicate the update of $MASDAWG(w)$. However, we can readily show the following lemma.

Lemma 8.5 *Suppose $b \in \Sigma$ and $u, bu \in Suffix(w)$. Let $x \in Factor(u)$ with $x \neq \varepsilon$. Let $y \in Factor(bu)$ with $y \neq \varepsilon$. Assume x and y are the representatives of $[x]^R_u$ and $[y]^R_{bu}$, respectively. Suppose $\langle u, [x]\rangle \sim_w \langle bu, [y]\rangle$. Let $a \in \Sigma$, and let z be the longest repeated suffix of bua. Suppose $z \in [y]^R_{bu}$. If $|z| < |y|$, then z is also the longest repeated suffix of*

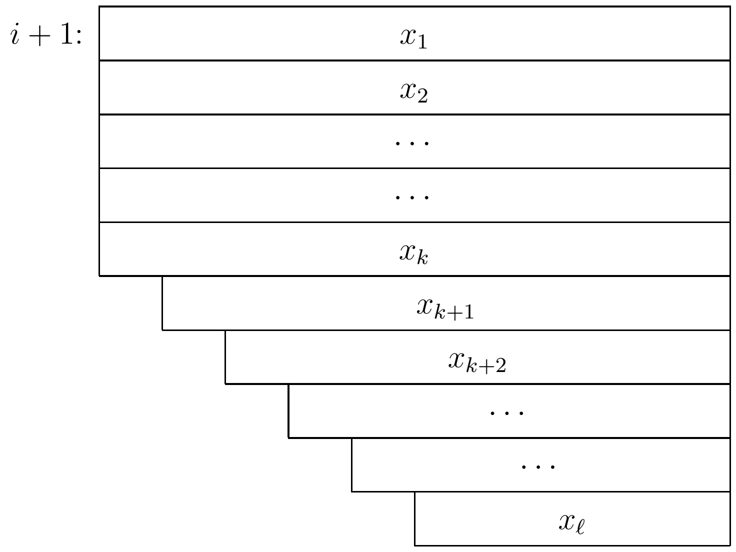


Figure 8.4: The representatives x_j of $[x_j]^R_{w[i+j:]}$ such that the nodes $\langle w[i+j:], [x_j]\rangle$ of the naive *ASDAWG*(w) are merged into one node of *MASDAWG*(w).

ua, and $z \in [x]^R_u$. If $|z| = |y|$, then x is a repeated suffix of ua (not necessarily to be the longest).

The next lemma characterizes the node separations that occur during the update of *MASDAWG*(w) to *MASDAWG*(wa).

Lemma 8.6 *Consider the node of MASDAWG(w) stated in Lemma 8.4 (see Fig. 8.4). Let z be the longest repeated suffix of $w[i+j:]a$. Suppose $z \in [x_j]^R_{w[i+j:]}$.*

1. *When $|z| = |x_k|$: Node separation occurs in none of the DAWGs for the strings $w[i+j:], \dots, w[i+\ell:]$.*
2. *When $|z| < |x_k|$: Let t be the maximum integer such that z is a proper suffix of x_t. Node separation occurs in each of the DAWGs for the strings $w[i+j:], \dots, w[i+t:]$. That is, for each $j = 1, \dots, t$, the node $[x_j]^R_{w[i+j:]}$ of DAWG$(w[i+j:])$ is separated into $[x_j]^R_{w[i+j:]a}$ and $[z]^R_{w[i+j:]a}$ inside DAWG$(w[i+j:]a)$. The nodes $\langle w[i+j:]a, [x_1]\rangle, \dots, \langle w[i+\ell:]a, [x_\ell]\rangle$ are equivalent under $\sim_{wa}$, and the new nodes $\langle w[i+j:]a, [z]\rangle, \dots, \langle w[i+t:], [z]\rangle$ are also equivalent under $\sim_{wa}$.*

The node separations of DAWGs characterized in the above lemma lead to a node separation in the update of *MASDAWG*(w) to *MASDAWG*(wa). It simultaneously performs the node separations within each DAWG caused by the common z. (For the same z, we can take j as small as possible.)

The remaining problem to be overcome is that there is another kind of node separation in the update of $MASDAWG(w)$.

Lemma 8.7 *In the update of $MASDAWG(w)$ to $MASDAWG(wa)$, node separation of the following types may occur, where $w \in \Sigma^*$ and $a \in \Sigma$.*

1. *When $w[i+1:]$ is of the form $b^{\ell+1}s$ such that $w[i] \neq b$ or $i = 0$, $\ell \geq 1$, and s in Σ^* does not begin with b and does not contain an occurrence of b^{ℓ}:*
   Let d be the largest integer such that s contains an occurrence of b^d. $MASDAWG(w)$ has a node consisting of
   $$\langle w[i+j+1:], [b^{d+k}]\rangle, \langle w[i+j+2:], [b^{d+k-1}]\rangle, \ldots, \langle w[i+j+k], [b^{d+1}]\rangle,$$
   where $k = \ell - (d+j) + 1$, for each $j = 0, 1, \ldots, d$. If $|s| > 0$, s ends with b^d, and $a = b$, then the node is separated into two nodes, one of which consists of
   $$\langle w[i+j+1:]a, [b^{d+k}]\rangle, \langle w[i+j+2:]a, [b^{d+k-1}]\rangle, \ldots, \langle w[i+j+k-1]a, [b^{d+2}]\rangle,$$
   and the other consists only of $\langle w[i+j+k:]a, [b^{d+1}]\rangle$.

2. *When $w[i+1:]$ is of the form b^{ℓ} with $\ell \geq 1$ such that $w[i] \neq b$ or $i = 0$: $MASDAWG(w)$ has a node consisting of*
   $$\langle b^{\ell}, [b^j]\rangle, \langle b^{\ell-1}, [b^{j-1}]\rangle, \ldots, \langle b^{\ell-j}, [\varepsilon]\rangle,$$
   for each $j = 1, \ldots, \ell$. Whenever $b \neq a$, the node is separated into two nodes, one of which consists of
   $$\langle b^{\ell}a, [b^j]\rangle, \langle b^{\ell-1}a, [b^{j-1}]\rangle, \ldots, \langle b^{\ell-j+1}a, [b]\rangle,$$
   and the other consists only of $\langle b^{\ell-j}a, [\varepsilon]\rangle$,

For an example of the first case of the above lemma, consider the update of $MASDAWG(w)$ to $MASDAWG(wb)$ for $w = bbbbbab$, which can be found in Fig. 8.5 and Fig. 8.6.

It should be emphasized that in the node separation mentioned in the above lemma no node separation occurs inside a DAWG. This kind of node separation can also be performed during the suffix link traversal started at the sink node, although the detail is omitted in this chapter.

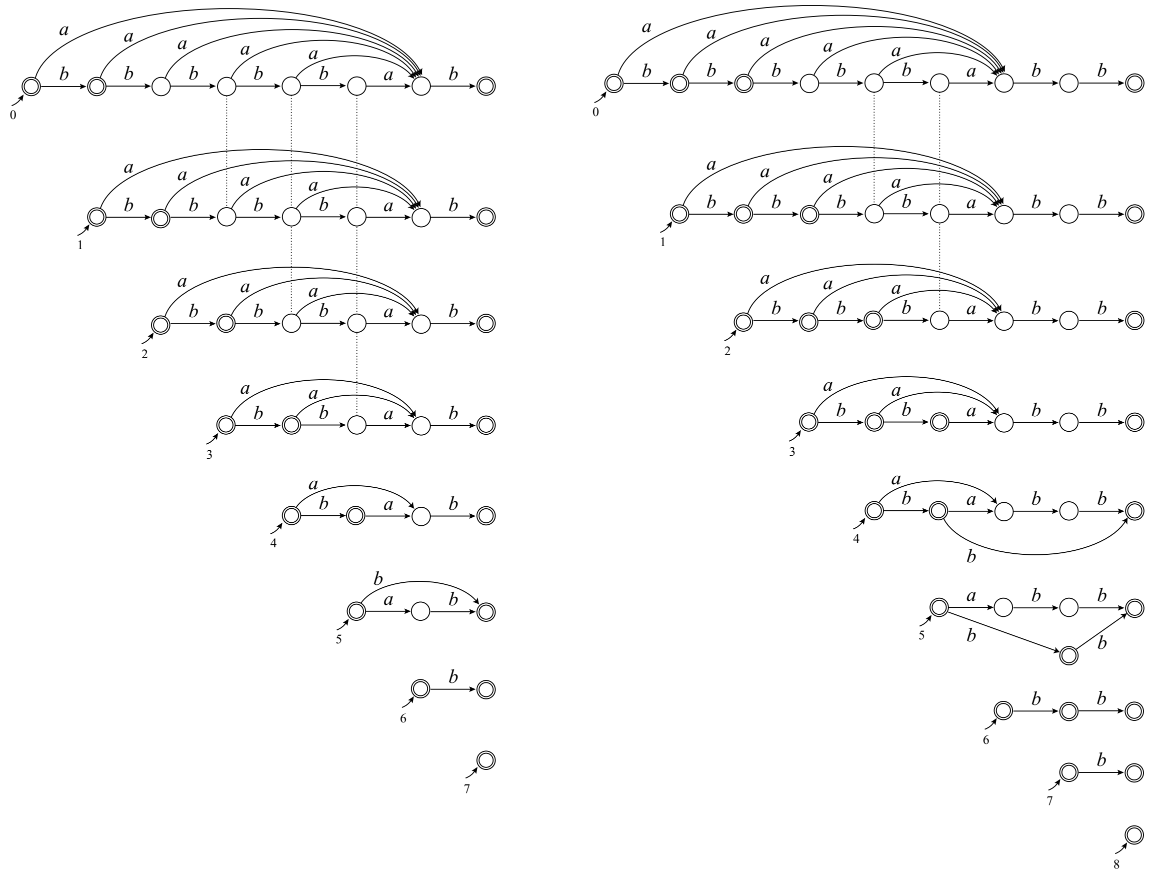


Figure 8.5: The naive *ASDAWG*(*bbbbbab*) on the left, and the naive *ASDAWG*(w) on the right. The nodes connected by the broken lines are equivalent due to Case 3. Recall the value of "d" mentioned in Lemma 8.7. In string *bbbbbab* the value of d is 1, whereas in string *bbbbbabb* $d = 2$ since the new b is added afterward.

8.4 Applications

In this section we show some applications to which the data structure *ASDAWG* and its variants effectively contribute.

8.4.1 Finding Beginning-Sensitive Patterns

Definition 8.2 (Beginning-Sensitive Pattern) *A* beginning-sensitive pattern *(a* BS-pattern *for short) is a pair $\langle p, i\rangle$ where p is a string in Σ^* and i is a positive integer.*

Definition 8.3 (BS-Pattern Matching Problem)
Instance*: a text w and a BS-pattern $\langle p, i\rangle$.*

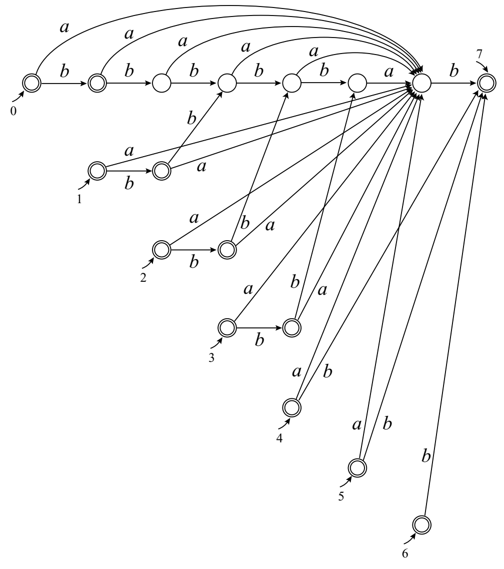
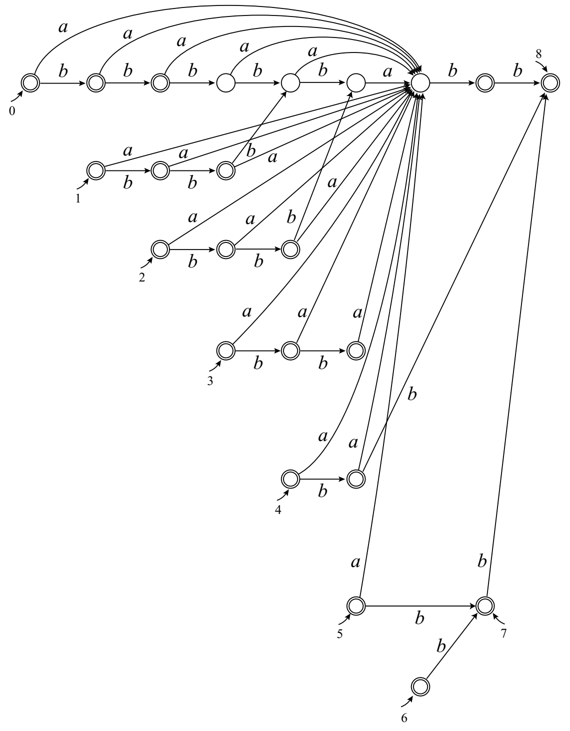



Figure 8.6: *MASDAWG*(*bbbbbab*) is on the left, and *MASDAWG*(*bbbbbabb*) is on the right. Compare the update of *MASDAWG*(*bbbbbab*) to *MASDAWG*(*bbbbbabb*) with that of the naive *ASDAWG*(*bbbbbab*) to the naive *ASDAWG*(*bbbbbabb*) shown in Fig. 8.5.

Determine: *whether p is a substring of $w[i:]$.*

This is a natural extension of the substring pattern matching problem with $i = 1$. The BS-pattern matching problem is solvable in $O(|p|)$ for an arbitrary pair $\langle p, i\rangle$, by using *ASDAWG*(w). For a given text w, we construct *MASDAWG*(w) with the on-line algorithm proposed in Section 8.3. For a BS-pattern $\langle p, i\rangle$, if $i > |w|$, the BS-pattern never matches w. Otherwise, we start with the i-th initial state of *MASDAWG*(w) and examine whether or not the string p is recognized.

8.4.2 Pattern Matching within a Specific Region

Definition 8.4 (Region-Sensitive Pattern) *A* region-sensitive pattern *(an* RS-pattern *for short) is a triple $\langle p, (i, j)\rangle$ where p is a string in Σ^* and i, j are positive integers.*

Definition 8.5 (RS-Pattern Matching Problem)
Instance: *a text w and an RS-pattern $\langle p, (i, j)\rangle$.*

Determine: *whether p occurs within the region $w[i:j]$ in the text w.*

This is a natural extension of the BS-pattern matching problem in which $j = |w|$. For a given text w, we construct $\mathit{MASDAWG}(w)$. It is a trivial fact that, while constructing $\mathit{MASDAWG}(w)$, the on-line algorithm is able to mark each state with the integer for the position of the *right most occurrence* of the string corresponding to the state. For an RS-pattern $\langle p, (i, j)\rangle$, if $i > |w|$, the RS-pattern never matches w. Otherwise, we start with the i-th initial state of $\mathit{MASDAWG}(w)$ and examine whether or not the string p is recognized. If it is recognized, we compare j with the integer k stored in the state at which p finally arrived. Then: If $j \leq k$, YES; Otherwise, NO. Obviously, the problem can be solved in $O(|p|)$ time.

8.4.3 Finding Variable-Length-Don't-Care's Patterns

We remark that $\mathit{MASDAWG}(w)$ can also be a powerful structure to find a *variable-length-don't-care's patterns*. The detail will be treated in Section 9.5 with strong possibility that $\mathit{MASDAWG}(w)$ can contribute to Knowledge Discovery and Data Mining.

Chapter 9

Finding Patterns to Best Separate Two Sets of Strings

9.1 Introduction

In these days, a lot of text data or sequential data are available, and it is quite important to discover useful rules from these data. Finding a *good rule* to separate two given sets, often referred as *positive examples* and *negative examples*, is a critical task in Knowledge Discovery and Data Mining.

In [22], Hirao et al. considered *subsequence patterns* as rules. A subsequence pattern s *matches* a string t if s can be obtained by deleting zero or more characters from t. They introduced a practical algorithm to find the best subsequence pattern that separates positive examples from negative examples, and showed some experimental results. A drawback of subsequence patterns is that they are not suitable for classifying *long* strings over *small* alphabet, since a short subsequence pattern matches almost all long strings.

In this chapter, we consider *episode patterns*, which were originally introduced by Mannila et al. [42]. An episode pattern $\langle v, k\rangle$, where v is a string and k is an integer, *matches* a string t if v is a subsequence for some substring u of t with $|u| \leq k$. Episode patterns are generalization of subsequence patterns since a subsequence pattern v is equivalent to the episode pattern $\langle v, \infty\rangle$. We give a practical solution to find the best episode pattern which separates one set of strings from the other set of strings. We propose a practical implementation of exact search algorithm that practically avoids exhaustive search. The key idea is to introduce some heuristics to reduce the search space based on the combinatorial properties of episode patterns, and to utilize an efficient data structure that helps to determine whether an episode pattern matches with a fixed string, at the cost of preprocessing time and space requirement to construct it. The result was published in [23], and also reported in [49].

At the end of the chapter, we also consider the *variable-length-don't-care's patterns*. Let $\Pi = (\Sigma \cup \{\star\})^*$, where $\star$ is a *wildcard* that matches any string. A pattern $q \in \Pi$ such as $q = a\star ba\star c$ is called a *variable-length-don't-care's pattern* (*VLDC-pattern*), where $a, b \in \Sigma$. The *language* $L(q)$ of a pattern $q \in \Pi$ is the set of strings obtained by replacing $\star$'s in q with strings. For example, $L(a\star ba\star c) = \{aubavc \mid u, v \in \Sigma^*\}$. This language corresponds to a class of the *pattern languages* proposed by Angluin [1]. We have strongly believed that the VLDC-patterns can be quite good rules to separate given two sets of strings. We declare that the smallest automaton to recognize all possible VLDC-patterns matching a text w is a variant of $MASDAWG(w)$, introduced in Chapter 8.

9.2 Preliminaries

9.2.1 Notation

Let $\mathcal{N}$ be the set of integers. Throughout this chapter, we use the word *substring* in the same meaning as the word *factor*. We say that a string v is a *subsequence* of a string w if v can be obtained by removing zero or more characters from w. We write as $v \preceq_{\rm str} w$ if v is a substring of w, and as $v \preceq_{\rm seq} w$ if v is a subsequence of w.

For a string v, we define the *substring language* $L^{\rm str}(v)$ and *subsequence language* $L^{\rm seq}(v)$ as follows:

$$\begin{aligned} L^{\rm str}(v) &= \{w \in \Sigma^* \mid v \preceq_{\rm str} w\}, \\ L^{\rm seq}(v) &= \{w \in \Sigma^* \mid v \preceq_{\rm seq} w\}. \end{aligned}$$

An *episode pattern* is a pair of a string v and an integer k, and we define the *episode language* $L^{\rm eps}(\langle v, k\rangle)$ by

$$L^{\rm eps}(\langle v, k\rangle) = \{w \in \Sigma^* \mid {}^{\exists}u \preceq_{\rm str} w \text{ such that } v \preceq_{\rm seq} u \text{ and } |u| \leq k\}.$$

The following lemma is obvious from the definitions.

Lemma 9.1 (Hirao et al. [22]) *For any strings $v, w \in \Sigma^*$,*

1. *if v is a prefix of w, then $v \preceq_{str} w$,*

2. *if v is a suffix of w, then $v \preceq_{str} w$,*

3. *if $v \preceq_{str} w$ then $v \preceq_{seq} w$,*

4. *$v \preceq_{str} w$ if and only if $L^{str}(v) \supseteq L^{str}(w)$,*

5. *$v \preceq_{seq} w$ if and only if $L^{seq}(v) \supseteq L^{seq}(w)$.*

9.2.2 Formulation of the Problem

Let *good* be a function from $\Sigma^* \times 2^{\Sigma^*} \times 2^{\Sigma^*}$ to the set of real numbers. We formulate the problem to be solved as follows.

Definition 9.1 (Finding the best pattern according to *good*)
Input: *Two sets $S, T \subseteq \Sigma^*$ of strings.*
Output: *A string $w \in \Sigma^*$ that maximizes the value $good(w, S, T)$.*

Intuitively, the value $good(w, S, T)$ expresses the goodness w to separate S, T. The definition of *good* varies for each application. For examples, the χ^2 values, entropy information gain, and gini index can be used. Essentially, these statistical measures are defined by the numbers of strings that satisfy the rule specified by w. Any of the above examples of the measures can be described in the following form:

$$\begin{aligned} good(w, S, T) &= f(x_w, y_w, |S|, |T|), \text{ where} \\ x_w &= |S \cap L^{\text{eps}}(w)|, \\ y_w &= |T \cap L^{\text{eps}}(w)|. \end{aligned}$$

When the sets S and T are fixed, the values $|S|$ and $|T|$ become constants. Thus, we abbreviate the function to $f(x, y)$ in the sequel.

Since the function $good(w, S, T)$ expresses the goodness of a episode pattern w to distinguish two sets, it is natural to assume that the function f satisfies the *conicality*, defined as follows.

Definition 9.2 *We say that a function $f(x, y)$ is* conic *if*

- *for any $0 \le y \le y_{\max}$, there exists an x_1 such that*

  - $f(x, y) \geq f(x', y)$ *for any* $0 \leq x < x' \leq x_1$, *and*
  - $f(x, y) \leq f(x', y)$ *for any* $x_1 \leq x < x' \leq x_{\max}$.

- *for any* $0 \leq x \leq x_{\max}$, *there exists a* y_1 *such that*
  - $f(x, y) \geq f(x, y')$ *for any* $0 \leq y < y' \leq y_1$, *and*
  - $f(x, y) \leq f(x, y')$ *for any* $y_1 \leq y < y' \leq y_{\max}$.

We assume that f is conic and can be evaluated in constant time in the sequel. The optimization problem to be tackled follow.

Definition 9.3 (Finding the best substring pattern according to f)
Input: *Two sets* $S, T \subseteq \Sigma^*$ *of strings.*
Output: *A string* v *that maximizes the value* $f(x_v, y_v)$, *where* $x_v = |S \cap L^{str}(v)|$ *and* $y_v = |T \cap L^{str}(v)|$.

Definition 9.4 (Finding the best subsequence pattern according to f)
Input: *Two sets* $S, T \subseteq \Sigma^*$ *of strings.*
Output: *A string* v *that maximizes the value* $f(x_v, y_v)$, *where* $x_v = |S \cap L^{seq}(v)|$ *and* $y_v = |T \cap L^{seq}(v)|$.

Definition 9.5 (Finding the best episode pattern according to f)
Input: *Two sets* $S, T \subseteq \Sigma^*$ *of strings.*
Output: *An episode pattern wpat that maximizes the value* $f(x_{\langle v,k\rangle}, y_{\langle v,k\rangle})$, *where* $x_{\langle v,k\rangle} = |S \cap L^{eps}(\langle v, k\rangle)|$ *and* $y_{\langle v,k\rangle} = |T \cap L^{eps}(\langle v, k\rangle)|$.

We remark that the first problem can be solved in linear time [22], while the latter two are NP-hard [44].

We review the basic idea of our algorithms. Fig. 9.1 shows a naive algorithm which exhaustively examines and evaluate all possible patterns one by one, and returns the best pattern that gives the maximum value. The most time-consuming part is obviously the lines 3 and 4. In order to reduce the search time, we should (1) reduce the possible patterns in line 2 *dynamically* by using some appropriate pruning method, and (2) speed up the computation of $|S \cap L(\pi)|$ and $|T \cap L(\pi)|$ for each π. In Section 9.3, we deal with (1), and in Section 9.4, we treat (2).

```
pattern FindMaxPattern(StringSet $S$, $T$)
1  $maxVal = -\infty$;
2  for all possible pattern $\pi$ do
3      $x = |S \cap L(\pi)|$;
4      $y = |T \cap L(\pi)|$;
5      $val = f(x, y)$;
6      if $val > maxVal$ then
7          $maxVal = val$;
8          $maxSeq = \pi$;
9  return $maxSeq$;
```

Figure 9.1: Exhaustive search algorithm.

9.3 Pruning Heuristics

In this section, we introduce some pruning heuristics, inspired by Morishita and Sese [45].

For a function $f(x, y)$, we denote $F(x, y) = \max\{f(x, y), f(x, 0), f(0, y), f(0, 0)\}$. From the definition of conic function, we can prove the following lemma.

Lemma 9.2 *For any patterns v and w with $L(v) \supseteq L(w)$, we have*

$$f(x_w, y_w) \leq F(x_v, y_v).$$

9.3.1 For Subsequence Patterns

We consider finding subsequence pattern in this subsection. By Lemma 9.1 (5) and Lemma 9.2, we have the following lemma.

Lemma 9.3 (Hirao et al. [22]) *For any strings $v, w \in \Sigma^*$ with $v \preceq_{seq} w$, we have*

$$f(x_w, y_w) \leq F(x_v, y_v).$$

In Fig. 9.2, we show our algorithm to find the best subsequence pattern from given two sets of strings, according to the function f. Optionally, we can specify the maximum length of subsequences. We use the following data structures in the algorithm.

StringSet Maintain a set S of strings.

- **int** *numOfSubseq*(**string** *seq*) : return the cardinality of the set $\{w \in S \mid seq \preceq_{\mathrm{seq}} w\}$.

```
string FindMaxSubsequence(StringSet S, T, int maxLength = ∞)
1  string prefix, seq, maxSeq;
2  double upperBound = ∞, maxVal = −∞, val;
3  int x, y;
4  PriorityQueue queue;    /* Best First Search*/
5  queue.push("", ∞);
6  while not queue.empty() do
7      (prefix, upperBound) = queue.pop();
8      if upperBound < maxVal then break;
9      foreach $c \in \Sigma$ do
10         seq= prefix+ c;    /* string concatenation */
11         x = S.numOfSubseq(seq);
12         y = T.numOfSubseq(seq);
13         val = $f(x, y)$;
14         if val > maxVal then
15             maxVal = val;
16             maxSeq = seq;
17*        upperBound = $F(x, y)$;
18         if |seq| < maxLength then
19             queue.push(seq, upperBound);
20 return maxSeq;
```

Figure 9.2: Algorithm *FindMaxSubsequence*.

PriorityQueue Maintain strings with their priorities.

- **bool** *empty*() : return **true** if the queue is empty.
- **void** *push*(**string** w, **double** *priority*) : push a string w into the queue with priority *priority*.
- **(string, double)** *pop*() : pop and return a pair (*string*, *priority*), where *priority* is the highest in the queue.

The next theorem guarantees the completeness of the algorithm.

Theorem 9.1 (Hirao et al. [22]) *Let S and T be sets of strings, and ℓ be a positive integer. The algorithm FindMaxSubsequence(S, T, ℓ) will return a string w that maximizes the value $f(x_v, y_v)$ among the strings of length at most ℓ, where $x_v = |S \cap L^{seq}(s)|$ and $y_s = |T \cap L^{seq}(s)|$.*

9.3.2 For Episode Patterns

We now show a practical algorithm to find the best episode patterns. We should remark that the search space of episode patterns is $\Sigma^* \times \mathcal{N}$, while the search space of subsequence patterns was Σ^*. A straight-forward approach based on the last subsection might be as follows. First we observe that the algorithm *FindMaxSubsequence* in Fig. 9.2 can be easily modified to find the best episode pattern $\langle v, k\rangle$ *for any fixed threshold* k: we have only to replace the lines 11 and 12 so that they compute the numbers of strings in S and T that match with the episode pattern $\langle seq, k\rangle$, respectively. Thus, for each possible threshold value k, repeat his algorithm, and get the maximum. A short consideration reveals that we have only to consider the threshold values up to l, that is the length of the longest string in given S and T.

However, here we give a more efficient solution. Let us consider the following problem, that is a subproblem of *finding the best episode pattern* in Definition 9.5.

Definition 9.6 (Finding the best threshold value)
Input: *Two sets* $S, T \subseteq \Sigma^*$ *of strings, and a string* $v \in \Sigma^*$.
Output: *Integer* k *that maximizes the value* $f(x_{\langle v,k\rangle}, y_{\langle v,k\rangle})$, *where* $x_{\langle v,k\rangle} = |S \cap L^{eps}(\langle v, k\rangle)|$ *and* $y_{\langle v,k\rangle} = |T \cap L^{eps}(\langle v, k\rangle)|$.

The next lemma give a basic containment of episode pattern languages.

Lemma 9.4 (Hirao et al. [23]) *For any two episode patterns* $\langle v, l\rangle$ *and* $\langle w, k\rangle$, *if* $v \preceq_{seq} w$ *and* $l \geq k$ *then* $L^{eps}(\langle v, l\rangle) \supseteq L^{eps}(\langle w, k\rangle)$.

By Lemma 9.2 and 9.4, we have the next lemma.

Lemma 9.5 (Hirao et al. [23]) *For any two episode patterns* $\langle v, l\rangle$ *and* $\langle w, k\rangle$, *if* $v \preceq_{seq} w$ *and* $l \geq k$ *then* $f(x_{\langle w,k\rangle}, y_{\langle w,k\rangle}) \leq F(x_{\langle v,l\rangle}, y_{\langle v,l\rangle})$.

For strings $v, s \in \Sigma^*$, we define the *threshold value* θ of v for s by $\theta = min\{k \in \mathcal{N} \mid s \in L^{\mathrm{eps}}(\langle v, k\rangle)\}$. If no such value, let $\theta = \infty$. Note that $s \notin L^{\mathrm{eps}}(\langle v, k\rangle)$ for any $k < \theta$, and $s \in L^{\mathrm{eps}}(\langle v, k\rangle)$ for any $k \geq \theta$. For a set S of strings and a string v, let us denote by $\Theta_{S,v}$ the set of threshold values of v for some $s \in S$.

A key observation is that a best threshold value for given $S, T \subseteq \Sigma^*$ and a string $v \in \Sigma^*$ can be found in $\Theta_{S,v} \cup \Theta_{T,v}$ without loss of generality. Thus we can restrict the search space of the best threshold values to $\Theta_{S,v} \cup \Theta_{T,v}$.

From now on, we consider the numerical sequence $\{x_{\langle v,k\rangle}\}_{k=0}^{\infty}$. (We will treat $\{y_{\langle v,k\rangle}\}_{k=0}^{\infty}$ in the same way.) It clearly follows from Lemma 9.4 that the sequence is non-decreasing. Remark that $0 \leq x_{\langle v,k\rangle} \leq |S|$ for any k. Moreover, $x_{\langle v,l\rangle} = x_{\langle v,l+1\rangle} = x_{\langle v,l+2\rangle} = \cdots$, where l is the length of the longest string in S. Hence, we can represent $\{x_{\langle v,k\rangle}\}_{k=0}^{\infty}$ with a list having at most $\min\{|S|, l\}$ elements. We call this list *a compact representation of the sequence* $\{x_{\langle v,k\rangle}\}_{k=0}^{\infty}$ (*CRS*, for short).

We show how to compute CRS for each v and a fixed S. Observe that $x_{\langle v,k\rangle}$ increases only at the threshold values in $\Theta_{S,v}$. By computing a sorted list of all threshold values in $\Theta_{S,v}$, we can construct the CRS of $\{x_{\langle v,k\rangle}\}_{k=0}^{\infty}$. If using the counting sort, we can compute the CRS for any $v \in \Sigma^*$ in $O(|S|ml + |S|) = O(||S||m)$ time, where $m = |v|$.

We emphasize that the time complexity of computing the CRS of $\{x_{\langle v,k\rangle}\}_{k=0}^{\infty}$ is the same as that of computing $x_{\langle v,k\rangle}$ for a single k $(0 \leq k \leq \infty)$, by our method.

After constructing CRSs $\bar{x}$ of $\{x_{\langle v,k\rangle}\}_{k=0}^{\infty}$ and $\bar{y}$ of $\{y_{\langle v,k\rangle}\}_{k=0}^{\infty}$, we can compute the best threshold value in $O(|\bar{x}| + |\bar{y}|)$ time. Thus we have the following, which gives an efficient solution to the finding the best threshold value problem.

Lemma 9.6 *Given $S, T \subseteq \Sigma^*$ and $v \in \Sigma^*$, we can find the best threshold value in $O(\,(||S|| + ||T||)\cdot|v|\,)$ time.*

By substituting this procedure into the algorithm *FindMaxSubsequence*, we get an algorithm to find a best episode pattern from given two sets of strings, according to the function f, shown in Fig. 9.3. We add a method *crs*(v) to the data structure **StringSet** that returns CRS of $\{x_{\langle v,k\rangle}\}_{k=0}^{\infty}$, as mentioned above.

By Lemma 9.5, we can use the value *upperBound* $= F(x_{v,\infty}, y_{v,\infty})$ to prune branches in the search tree computed at line 19 marked by (*). We emphasize that the value $F(x_{\langle v,k\rangle}, y_{\langle v,k\rangle})$ is insufficient as *upperBound*. Note also that $x_{\langle v,\infty\rangle}$ and $y_{\langle v,\infty\rangle}$ can be extracted from $\bar{x}$ and $\bar{y}$ in constant time, respectively. The next theorem guarantees the completeness of the algorithm.

Theorem 9.2 *Let S and T be sets of strings, and ℓ be a positive integer. The algorithm FindBestEpisode(S, T, ℓ) will return an episode pattern that maximizes $f(x_{\langle v,k\rangle}, y_{\langle v,k\rangle})$, with $x_{\langle v,k\rangle} = |S \cap L^{eps}(\langle v,k\rangle)|$ and $y_{\langle v,k\rangle} = |T \cap L^{eps}(\langle v,k\rangle)|$, where v varies any string of length at most ℓ and k varies any integer.*

```
string $FindBestEpisode$(StringSet $S$, $T$, int $\ell$)
1   string $prefix$, $v$;
2   episodePattern $maxSeq$; /* pair of string and int */
3   double $upperBound = \infty$, $maxVal = -\infty$, $val$;
4   int $k'$;
5   CompactRepr $\bar{x}$, $\bar{y}$; /* CRS */
6   PriorityQueue $queue$;   /* Best First Search*/
7   $queue.push(\text{""}, \infty)$;
8   while not $queue.empty()$ do
9       $(prefix, upperBound) = queue.pop()$;
10      if $upperBound < maxVal$ then break;
11      foreach $c \in \Sigma$ do
12          $v = prefix + c$;   /* string concatenation */
13          $\bar{x} = S.crs(v)$;
14          $\bar{y} = T.crs(v)$;
15          $k' = \mathrm{argmax}_k\{f(x_{\langle v,k\rangle}, y_{\langle v,k\rangle})\}$ and $val = f(x_{\langle v,k'\rangle}, y_{\langle v,k'\rangle})$;
16          if $val > maxVal$ then
17              $maxVal = val$;
18              $maxEpisode = \langle v, k'\rangle$;
19(*)        $upperBound = F(x_{\langle v,\infty\rangle}, y_{\langle v,\infty\rangle})$;
20          if $upperBound > maxVal$ and $|v| < \ell$ then
21              $queue.push(v, upperBound)$;
22  return $maxEpisode$;
```

Figure 9.3: Algorithm *FindBestEpisode*.

9.4 Using Efficient Data Structures

In this chapter, we introduce a data structure efficient for the speedup of answering the queries.

9.4.1 Episode Directed Acyclic Subsequence Graphs

We now analyze the complexity of *episode pattern matching*. Given an episode pattern $\langle v, k\rangle$ and a string t, determine whether $t \in L^{\text{eps}}(\langle v, k\rangle)$ or not. This problem can be answered by filling up the edit distance table between v and t, where only insertion operation with cost one is allowed. It takes $\Theta(mn)$ time and space using a standard dynamic programming method, where $m = |v|$ and $n = |t|$. For a fixed string, automata-based approach is useful. Thereby we use the Episode Directed Acyclic Subsequence Graph (EDASG) for string t, which was recently introduced by Troíček in [53]. Hereafter,

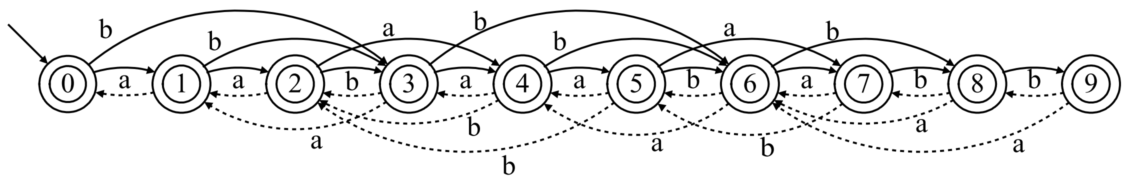


Figure 9.4: *EDASG*(t), where $t = aabaababb$. Solid arrows denote the forward edges, and broken arrows denote the backward edges. The number in each circle denotes the state number.

let *EDASG*(t) denote the EDASG for t. With the use of *EDASG*(t), episode pattern matching can be answered quickly in practice, although the worst case behavior is still $O(mn)$. EDASG(t) is also useful to compute the threshold value θ of given v for t quickly in practice. As an example, *EDASG*($aabaababb$) is shown in Fig. 9.4. When examining if an episode pattern $\langle abb, 4\rangle$ matches with t or not, we start from the initial state 0 and arrive at state 6, by traversing the forward edges spelling abb. It means that the shortest prefix of t that contains abb as a subsequences is $t[0:6] = aabaab$, where $t[i:j]$ denotes the substring $t_{i+1} \dots t_j$ of t. Moreover, the difference between the state numbers 6 and 0 corresponds to the length of matched substring $aabaab$ of t, that is, $6-0 = |aabaab|$. Since it exceeds the threshold 4, we move backwards spelling bba and reach state 1. It means that the shortest suffix of $t[0:6]$ that contains abb as a subsequence is $t[1:6] = abaab$. Since $6 - 1 > 4$, we have to examine other possibilities. It is not hard to see that we have only to consider the string $t[2:*]$. Thus we continue the same traversal started from state 2, that is the next state of state 1. By forward traversal spelling abb, we reach state 8, and then backward traversal spelling bba bring us to state 4. In this time, we found the matched substring $t[4:8] = abab$ which contains the subsequence abb, and the length $8 - 4 = 4$ satisfies the threshold. Therefore we report the occurrence and terminate the procedure.

It is not difficult to see that the EDASGs are useful to compute the threshold value of v for a fixed t. We have only to repeat the above forward and backward traversal up to the end, and return the minimum length of the matched substrings. Although the time complexity is still $\Theta(mn)$, practical behavior is usually better than using standard dynamic programming method.

9.5 Finding Variable-Length-Don't-Care's Patterns

In this chapter, we consider other kinds of patterns, called *variable-length-don't-care's patterns*. We here remark that the MASDAWG, introduced in Chapter 8, is a powerful structure to find patterns of these kinds.

The variable-length-don't-care's patterns are defined as follows.

Definition 9.7 (Variable-Length-Don't-Care's Patterns) *Let* $\Pi = (\Sigma \cup \{\star\})^*$, *where* $\star$ *is a* wildcard *that matches any string. An element* $q \in \Pi$ *is called a* variable-length-don't-care's pattern *(a* VLDC-pattern *for short).*

For instance, $\star a \star ab \star ba \star$ is a VLDC-pattern for $a, b \in \Sigma$. We say that a VLDC-pattern q matches a text string $w \in \Sigma^*$ if w can be obtained by replacing $\star$'s in q with some strings. In the running example, the VLDC-pattern $\star a \star ab \star ba \star$ matches text *abababbbaa* by replacing the $\star$'s with ab, b, b and a, respectively.

We write as $q \preceq_{\text{vldc}} w$ if a VLDC-pattern q matches w. For a VLDC-pattern q, we define the *VLDC-language* $L^{\text{vldc}}(q)$ as

$$L^{\text{vldc}}(q) = \{w \in \Sigma^* \mid q \preceq_{\text{vldc}} w\}.$$

Now we consider the following problem.

Definition 9.8 (Finding the best VLDC-pattern according to f)
Input: *Two sets* $S, T \subseteq \Sigma^*$ *of strings.*
Output: *A string* v *that maximizes the value* $f(x_v, y_v)$, *where* $x_v = |S \cap L^{vldc}(v)|$ *and* $y_v = |T \cap L^{vldc}(v)|$.

It is known that this problem is NP-hard [44].

Given a set S, T of strings, we have to examine whether or not every possible VLDC-pattern matches each string in S and T. Since the problem is NP-hard, we are forced to exponentially many candidate. Thus, when considering some efficient heuristics, we need some fast method for the examination. We address that the MASDAWGs are remarkably helpful for this. In fact, the smallest index structure capable of solving the following pattern matching problem in $O(|q|)$ time is a variant of *MASDAWG*(w).

Definition 9.9 (VLDC-Pattern Matching Problem)
Instance*: a text w and a VLDC-pattern q.*
Determine*: whether q matches w.*

The automaton recognizing all possible VLDC-patterns matching w is called the *wildcard ASDAWG* for w, and denoted by $WASDAWG(w)$. The automata were originally introduced in [25]. $WASDAWG(abbab)$ is displayed in Fig. 9.5. In $WASDAWG(w)$, a $\star$-transition is added between each state and the initial state of the "same layer" in $MASDAWG(w)$ (see also $MASDAWG(abbab)$ in Fig. 8.3). Note that there exist two additional states, one of which is a unique initial state of $WASDAWG(abbab)$. They are added in order that VLDC-patterns beginning with a can be recognized. For any $q \in \Pi$, the VLDC-pattern matching problem can be solved in $O(|q|)$ time, by using $WASDAWG(w)$.

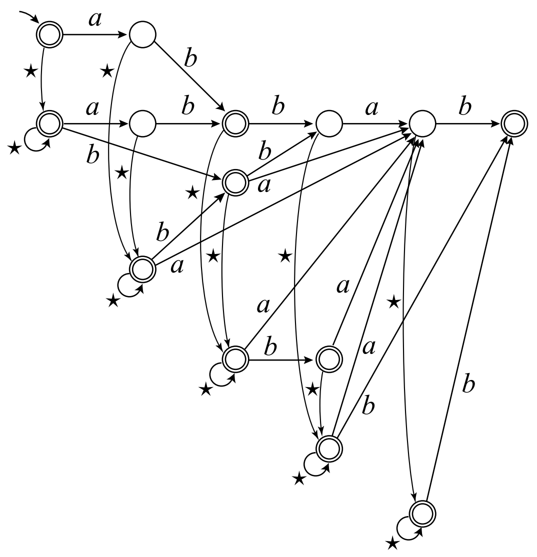


Figure 9.5: $WASDAWG(w)$ where $w = abbab$.

9.6 Concluding Remarks

This chapter was mainly devoted to the problem for finding *the best* episode patterns that effectively separate given two sets of strings.

Hamuro et al. [21] implemented the algorithm for finding the best subsequences, and reported a quite successful experiment on business data. In the meantime, Bannai et al. [6] and Iida et al. [27, 26] have installed our algorithms into the core of the decision tree generator in the BONSAI system [48]. They reported that they achieved significant results by using episode patterns as rules, on biological sequences.

On the other hand, we can easily extend our algorithm to enumerate *all* strings whose values of the objective function exceed the given threshold, since we essentially examine all strings, with effective pruning heuristics. Enumeration may be more preferable in the context of *text data mining* [10, 17, 57].

We also discussed the use of VLDC-patterns as rules for the separation. The invention of WASDAWGs is believed to be a clue to a practical algorithm to find the *best VLDC-patterns* that distinguish two sets of strings. The development of the pruning heuristic on using VLDC-patterns is our future work.

Acknowledgment

First and foremost, I wish to thank Prof. Ayumi Shinohara for his throughout supervision over the past two years. His enthusiastic suggestion and encouragement have truly been part of my motivation to tackle lots of difficult problems. He helped me every time I needed advice. Above all, I appreciate that he has offered me perfect environment and situation in which I have been able to focus only on my study. I add that the common experiences in the travels to Jerusalem (Israel), Laguna de San Rafael (Chile) and Washington DC (USA) are impressively nice, and unforgettable.

I would also like to thank Prof. Masayuki Takeda, my second supervisor. It was he who gave me the first opportunity to work for index structures, to which all topics in this thesis correspond. His attitude forward researching has been my model. The only thing remaining undone is that we have never been abroad together, though one of our 'prime' purposes is, as we are repeatedly saying, to go abroad and have fun there! I just hope we can do it this year.

I would also like to express my appreciation to Prof. Setsuo Arikawa, Prof. Hiroki Arimura and Dr. Hiroshi Sakamoto. They gave me fruitful advice on what to do and how to be as a researcher.

I also wish to thank my senior colleagues, Dr. Takuya Kida, Mr. Masahiro Hirao, Mr. Hiromasa Hoshino, and Mr. Tetsuya Matsumoto. They kindly helped me when I did not know much about the department and research, and they used to be my good model of presentation as well.

I am appreciative of my colleagues, Mr. Keisuke Iida, Mr. Katsuaki Taniguchi, Mr. Tatsuya Furuzono, Mr. Shuichi Mitarai, and Mr. Koichiro Yamamoto. They are my good friends, as well as give me some good stimulus on research aspect. Also, I am grateful to my friend Mr. Takaaki Miyachi, without whom I would never have become able to make use of English.

Last but not least, I thank my parents for their support.

Bibliography

[1] D. Angluin. Finding patterns common to a set of strings. *J. Comput. Sys. Sci.*, 21:46–62, 1980.

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

[3] A. Apostolico and Z. Galil. *Pattern Matching Algorithms.* Oxford University Press, New York, 1997.

[4] R. A. Baeza-Yates. Searching subsequences (note). *Theoretical Computer Science*, 78(2):363–376, Jan. 1991.

[5] M. Balík. Implementation of dawg. In *Proc. The Prague Stringology Club Workshop '98 (PSCW'98).* Czech Technical University, 1998.

[6] H. Bannai, K. Iida, A. Shinohara, M. Takeda, and S. Miyano. More speed and more pattern variations for knowledge discovery system BONSAI. In *Proc. the International Conference on Genome Informatics '01 (GIW'01)*, pages 454–455, 2001.

[7] 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.

[8] 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.

[9] D. Breslauer. The suffix tree of a tree and minimizing sequential transducers. *Theoretical Computer Science*, 191:131–144, 1998.

[10] A. Califano. SPLASH: Structural pattern localization analysis by sequential histograms. *Bioinformatics*, Feb. 1999.

[11] 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.

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

[13] M. Crochemore, C. S. Iliopoulos, T. Lecroq, and Y. J. Pinzon. Approximate string matching in musical sequences. In *Proc. The Prague Stringology Conference '01 (PSC'01)*, pages 26–36. Czech Technical University, Sept. 2001.

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

[15] M. Crochemore and Z. Troníček. Directed acyclic subsequence graph for multiple texts. Technical Report IGM-99-13, Institut Gaspard-Monge, June 1999.

[16] M. Crochemore and R. Vérin. On compact directed acyclic word graphs. In J. Mycielski, G. Rozenberg, and A. Salomaa, editors, *Structures in Logic and Computer Science*, volume 1261 of *Lecture Notes in Computer Science*, pages 192–211. Springer-Verlag, 1997.

[17] R. Feldman, Y. Aumann, A. Amir, A. Zilberstein, and W. Klosgen. Maximal association rules: A new tool for mining for keyword co-occurrences in document collections. In *Proc. of the 3rd International Conference on Knowledge Discovery and Data Mining*, pages 167–170. AAAI Press, Aug. 1997.

[18] 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.

[19] 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.

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

[21] Y. Hamuro, H. Kawata, N. Katoh, and K. Yada. A machine learning algorithm for analyzing string patterns helps to discover simple and interpretable business rules from purchase history. In S. Arikawa and A. Shinohara, editors, *Progress in Discovery Science*, Lecture Notes in Computer Science. Springer-Verlag, 2002.

[22] M. Hirao, H. Hoshino, A. Shinohara, M. Takeda, and S. Arikawa. A practical algorithm to find the best subsequence patterns. In S. Arikawa and S. Morishita, editors, *Proc. The Third International Conference on Discovery Science*, volume 1967 of *Lecture Notes in Artificial Intelligence*, pages 141–154. Springer-Verlag, 2000.

[23] M. Hirao, S. Inenaga, A. Shinohara, M. Takeda, and S. Arikawa. A practical algorithm to find the best episode patterns. In K. P. Jantke and A. Shinohara, editors, *Proc. The Fourth International Conference on Discovery Science*, volume 2226 of *Lecture Notes in Artificial Intelligence*, pages 435–440. Springer-Verlag, 2001.

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

[25] H. Hoshino. Construction of data structure for knowledge discovery on text data. Master's thesis, Department of Informatics, Kyushu University, Feb. 2001. (in Japanese).

[26] K. Iida. Knowledge discovery on molecular biological data. Master's thesis, Department of Informatics, Kyushu University, Feb. 2002. (in Japanese).

[27] K. Iida, H. Bannai, A. Shinohara, M. Takeda, and S. Miyano. Extension and speed up of knowledge discovery system BONSAI. In *Proc. the 7th annual Pacific Symposium on Biocomputing (PSB'02)*, page 100, 2002.

[28] S. Inenaga. Bidirectional on-line linear-time construction of suffix trees and directed acyclic word graphs. Technical Report DOI-TR-CS-203, Department of Informatics, Kyushu University, Feb. 2002.

[29] 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, Sept. 2001.

[30] 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.

[31] 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, Aug. 2001.

[32] 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 A. Amir and G. M. Landau, editors, *Proc. 12th Annual Symposium on Combinatorial Pattern Matching (CPM'01)*, volume 2089 of *Lecture Notes in Computer Science*, pages 169–180. Springer-Verlag, 2001.

[33] S. Inenaga, M. Takeda, A. Shinohara, H. Hoshino, and S. Arikawa. The minimum dawg for all suffixes of a string and its applications. Technical Report DOI-TR-CS-204, Department of Informatics, Kyushu University, Feb. 2002.

[34] T. Kadota, M. Hirao, A. Ishino, M. Takeda, A. Shinohara, and F. Matsuo. Musical sequence comparison for melodic and rhythmic similarities. In *Proc. of 8th International Symposium on String Processing and Information Retrieval (SPIRE'01)*, pages 111–122. IEEE Computer Society, 2001.

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

[36] D. E. Knuth, J. H. Morris, and V. R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(2):323–350, 1977.

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

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

[39] M. G. Maaß. Linear bidirectional on-line construction of affix trees. In R. Giancarlo and D. Sankoff, editors, *Proc. 11th Annual Symposium on Combinatorial Pattern Matching (CPM'00)*, volume 1848 of *Lecture Notes in Computer Science*, pages 320–334. Springer-Verlag, 2000.

[40] V. Mäkinen. Compact suffix array. In R. Giancarlo and D. Sankoff, editors, *Proc. 11th Annual Symposium on Combinatorial Pattern Matching (CPM'00)*, volume 1848 of *Lecture Notes in Computer Science*, pages 305–319. Springer-Verlag, 2000.

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

[42] H. Mannila, H. Toivonen, and A. I. Verkamo. Discovering frequent episode in sequences. In U. M. Fayyad and R. Uthurusamy, editors, *Proc. 1st International Conference on Knowledge Discovery and Data Mining*, pages 210–215. AAAI Press, Aug. 1995.

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

[44] S. Miyano, A. Shinohara, and T. Shinohara. Polynomial-time learning of elementary formal systems. *New Generation Computing*, 18:217–242, 2000.

[45] S. Morishita and J. Sese. Traversing itemset lattices with statistical metric pruning. In *Proc. of the 19th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 226–236. ACM Press, May 2000.

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

[47] 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.

[48] S. Shimozono, A. Shinohara, T. Shinohara, S. Miyano, S. Kuhara, and S. Arikawa. Knowledge acquisition from amino acid sequences by machine learning system BON-

SAI. *Transactions of Information Processing Society of Japan*, 35(10):2009–2018, Oct. 1994.

[49] A. Shinohara, M. Takeda, S. Arikawa, M. Hirao, H. Hoshino, and S. Inenaga. Practical algorithms to find the best string patterns. In S. Arikawa and A. Shinohara, editors, *Progress in Discovery Science*, volume 2281 of *Lecture Notes in Artificial Intelligence*, pages 308–318. Springer-Verlag, 2002.

[50] J. Stoye. Affixbäume. Master's thesis, Universität Bielefeld, may 1995. (in German).

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

[52] 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. *Theoretical Comput. Sci.*, 2002. (to appear).

[53] Z. Troníček. Episode matching. In A. Amir and G. M. Landau, editors, *Proc. 12th Annual Symposium on Combinatorial Pattern Matching (CPM'01)*, volume 2089 of *Lecture Notes in Computer Science*, pages 143–146. Springer-Verlag, 2001.

[54] E. Ukkonen. Approximate string matching over suffix trees. volume 684 of *Lecture Notes in Computer Science*, pages 228–242. Springer-Verlag, 1993.

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

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

[57] J. T. L. Wang, G.-W. Chirn, T. G. Marr, B. A. Shapiro, D. Shasha, and K. Zhang. Combinatorial pattern discovery for scientific data: Some preliminary results. In *Proc. of the 1994 ACM SIGMOD International Conference on Management of Data*, pages 115–125. ACM Press, May 1994.

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