String Processing Algorithms

Shunsuke Inenaga

March, 2003

Abstract

The thesis describes extensive studies on various algorithms for efficient string processing. Data available in/via computers are often of enormous size, and thus, it is significantly important and necessary to invent time- and space-efficient methods to process them. Most of such data are, in fact, stored and manipulated as *strings*.

String matching is most fundamental in string processing, where the problem is to examine whether or not a pattern string p occurs in a text string w. There are two cases to consider; p is fixed and w is flexible, and vise versa. In the former case, it is adequate to employ the noble algorithm by Knuth, Morris, and Pratt that solves the problem in $O(|w|)$ time using $O(|p|)$ space. The thesis, on the other hand, considers the latter case.

When w is fixed, it is natural, and ideal, to use a data structure that supports indices of w. Such a data structure is called an *index structure*. A linear-spaced index structure was first given by Weiner in 1973, named *suffix trees*. Suffix trees are regarded as a compaction of *suffix tries* that are a basic index structure requiring quadratic space. On the other hand minimizing suffix tries yields another type of index structure called *directed acyclic word graphs* (*DAWGs*), which was introduced by Blumer et al. in 1985. Moreover, by either minimizing suffix trees or compacting DAWGs gives us *compact directed acyclic word graphs* (*CDAWGs*). CDAWGs were also invented by Blumer et al. in 1987.

In the thesis we delve in those index structures, revealing their relationships in terms of equivalence classes on strings. After giving such theoretical characteristics of them, we explore ingenious algorithms related to those index structures for time- and space-efficient string processing in practice.

Particularly, we first introduce an *on-line* algorithm that *directly* constructs a CDAWG for a single string w in $O(|w|)$ time, and second, give its straightforward extension to a set S of strings whose running time is $O(\|S\|)$ where $\|S\|$ denotes the total length of the strings in S. A further, deeper analysis of the on-line algorithm gives us a *generalized* *algorithm* capable of building, in on-line, all of the four index structures, suffix tries, suffix trees, DAWGs, and CDAWGs. It clarifies an algorithmic unified view and reveals correspondence between algorithms and definitions of those structures.

The generalized algorithm is then improved so that a trie structure T representing a set S of strings can be given as an input. Except in case of constructing suffix tries that require quadratic space, the improved algorithm runs in $O(|T|)$ time, where $|T|$ is the number of nodes in the trie. The point is that we always have the inequality that $|T| \leq \|S\|$, and the more and longer prefixes the strings in S share, the less $|T|$ becomes.

Another context of using index structures could be employing them to support indices for *a part of a whole text w*, which is the so-called *sliding window mechanism*. When only a limited amount of memory is available, a sliding window is very useful and often applied actually; e.g., in the process of on-line data stream. Another application of sliding windows is a text compression scheme called *prediction by partial matching* (*PPM*). Larsson proposed a linear-time algorithm for building and maintaining suffix trees for a sliding window, and his algorithm contributed to reduce space-requirement in PPM. Since PPM is known to be the best text-compression scheme from the viewpoint of compression ratio, it is quite meaningful to pursue a reduction of space requirement. The thesis indeed gives a space-economical counterpart of Larsson's algorithm, i.e., an algorithm to construct and maintain CDAWGs for a sliding window is given here. Our algorithm runs in $O(|w|)$ time with $O(M)$ space like Larsson's algorithm, where M is the width of the sliding window.

Another challenging problem is to store indices of w and w^{rev} together, where w^{rev} is the reversal of w. The resulting structure is called a *bidirectional index structure*. Chen and Seiferas pointed out the fact that the suffix tree of w can share the same nodes with the DAWG of w^{rev}. In the thesis we give an on-line algorithm for building both the suffix tree of w and the DAWG of w^{rev}, in linear time and space. The most space-economical bidirectional index structure ever known is *symmetric compact directed acyclic word graphs* (*SCDAWGs*), invented by Blumer et al. in 1987. The thesis proposes an algorithm to construct SCDAWGs on-line, in linear time, and directly.

Updating index structures to both directions, to the right and left, is also a challenging and interesting problem indeed. Stoye invented a variant of suffix trees, called *affix trees*, being more suitable for bidirectional updating. Due to the fact that the size of affix trees is mostly larger than that of suffix tree, it could be said that Stoye's algorithm makes a sacrifice in space-requirement. As an improvement of this, the algorithm we propose here can update suffix trees themselves to both directions. Another good feature of affix trees is that they can also support indices of both w and w^{rev}. However, our algorithm can simultaneously update the suffix tree of w and the DAWG of w^{rev} to both directions, again saving memory space needed.

The above-mentioned topics were all about *the substring pattern matching problem*, and in the following we go on to consider more intricate pattern matching problems. We say that p is a *subsequence pattern* of w if p can be obtained by removing zero or more characters from w. *Episode patterns* are a generalized concept of subsequence patterns: a pair $\langle p, k \rangle$ is said to be an episode pattern of w if p is a subsequence of u such that u is a substring of w of length less than or equal to k.

The problem considered is as follows: "Given two sets of strings, find a pattern that is most abundant in one set and rarest in the other." The machine discovery system BONSAI developed by Shimozono et al. can find substring patterns that are best for the purpose, separation of two given sets of strings. In order to improve the BONSAI system Hirao et al. introduced a practical algorithm to find best subsequence patterns. The thesis explores a further extension of BONSAI which is capable of discovery of best episode patterns. The algorithm has been sped up in two ways; branch-and-bound heuristics and the use of index structures for episode pattern matching, called *episode directed acyclic subsequence graphs* (*EDASGs*).

Pattern matching with *wildcard* symbols has been studied extensively. A wildcard $\star$ is called a *variable-length-don't-care* (*VLDC*) symbol which matches any string, and a pattern containing $\star$'s is called a *VLDC pattern*. Since VLDC patterns are another type of generalization of subsequence patterns, it is highly desired to invent a fast and exact algorithm to discover VLDC patterns best for separating two given sets of strings. A similar branch-and-bound heuristics can be employed in this case as well, but some index structure to quickly solve the VLDC pattern matching problem was still needed. Therefor, we created *wildcard directed acyclic word graphs* (*WDAWGs*), with which the VLDC pattern matching problem is solvable in linear time with respect to the length of a given pattern. We give two algorithms to build WDAWGs in time linear in the output size. We then consider *a VLDC pattern within a window* that is composed of a pair $\langle q, k \rangle$, where q is a VLDC pattern and k is a threshold value bounding the length of the occurrence of q in w. A practical algorithm to discover the best pair $\langle q, k \rangle$ is also proposed in the thesis. We also execute some computational experiments to show the actual efficiency of our algorithms, by applying them to genomic sequences.

Acknowledgments

First and foremost, I wish to thank Prof. Ayumi Shinohara for his throughout supervision over the past three years. His enthusiastic encouragement totally supported me, and his apt advice guided me when I had difficulties in my research. Above all, I appreciate that he offered me a perfect environment and situation in which I was able to fully focus on my study.

I would also like to thank Prof. Masayuki Takeda, my second supervisor. It was he who gave me the first research topic. His attitude towards research has been my good model.

I would also like to express my appreciation to Prof. Setsuo Arikawa, Prof. Fumihiro Matsuo, and Prof. Kazuaki Murakami, who are the members of the committee of my thesis. I also thank all of those in Department of Informatics, Kyushu University, for their generous support.

This research was partly supported by JST (Japan Science and Technology). The results in the thesis were partially published in the proc. of CPM'01 and '02, the proc. of SPIRE'01 and '02, the proc. of PSC'01 and '02, the proc. of DS'01 and '02, the proc. of MFCS'02, the proc. of GIW'02, and Progress in Discovery Science. I am thankful for all editors, committees, anonymous referees, and publishers.

I would like to express my appreciation to Prof. Wojciech Rytter and Prof. Leszek Gasieniec. I enjoyed very much the discussion with them in my short visit to Department of Computer Science, University of Liverpool. One of the results of the thesis was published in the proc. of CPM'01 as a joint contribution with Giancarlo Mauri and Giulio Pavesi from University of Milan-Bicocca. I appreciate their comments for its extended version that is presented in the thesis. Also, I would express my thanks for all of those with whom I had fruitful discussions at conferences.

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

Contents

Chapter 1

Introduction

1.1 Background and Motivations

Due to rapid advance in information technology and global growth of computer networks, we can utilize a large amount of data today. This benefit, on the other hand, can turn out to be a serious matter that we have to make considerable effort in retrieving the information we really want and need from the enormous data available. *Strings* are the most fundamental form to store data in computers, and thus, string processing is the core of various applications in computer science. Lately, vast genomic sequences have been mostly determined, and they are about to become available to the public. Extracting important rules or knowledge from those string data sets, what is called *text mining*, has therefore attracted much attention and expectation. In order to process such huge data of giga- or even tera-bytes in a reasonable amount of time, space-economical data structures and fast algorithms are definitely necessary.

String matching is of central importance to string processing, and the problem is formalized as follows: "Given a text string w and a pattern string p, examine if p occurs in w or not." String p is said to be a *substring pattern* of w if $w = xpy$ with some strings x, y, and the substring pattern matching problem is most fundamental in string matching. The algorithm by Knuth, Morris and Pratt [45] is capable of solving this problem in $O(|w|)$ time using $O(|p|)$ space. Their algorithm constructs an automaton of size $O(|p|)$ that accepts any string which contains p as a substring, and runs the deterministic finite automaton over a given text w. Therefore, it is very useful and fast when p is fixed and w is flexible. In the opposite case where p is flexible and w is fixed, however, their algorithm is no longer adequate since whenever we get a new pattern p, we have to reconstruct the

DFA and run it over w which is often tremendously long. This implies that we should consider the use of some data structure that supports indices of w.

The most basic *index structure* for text strings is *suffix tries*. The suffix trie for w is a DFA that accepts all substrings of w, and thus it enables us to solve the substring pattern matching problem in $O(|p|)$ time. The crucial drawback is, however, that the suffix trie requires $O(|w|^2)$ space. It was Weiner who first succeeded improving the space requirement so as to be linear, by introducing *suffix trees* [77]. For any string w, its suffix tree can be obtained by removing any node having only one out-going edge from the suffix trie of w and concatenating the edges. We call this procedure *compaction* (see Figure 1.1). The suffix tree of w is regarded as a DFA accepting all substrings of w, whose transitions are labeled with strings. Weiner proposed an algorithm that directly constructs, for any string w, the suffix tree of w in $O(|w|)$ time. Later on, McCreight [55] gave a more-space economical algorithm for construction of suffix trees. Ukkonen [73] introduced an elegant *on-line* algorithm to build suffix trees, which in linear time processes a given string from left to right, one by one. Hence his algorithm allows us to extend the input string to the right, by adding new characters. Also, his algorithm is applicable to linear-time construction of the suffix tree for a set S of strings. Various applications of suffix trees can be found in the literature [16, 26, 47, 72].

Kosaraju introduced the suffix tree for a trie T that represents a set S of strings [46]. He also proposed an $O(|T| \log |T|)$ time algorithm to construct the suffix tree for T, where $|T|$ is the number of nodes in T. His trie T merges the suffixes of strings in S, and thus, the more and longer suffixes the strings in S share, the less $|T|$ becomes. Namely, we always have the inequality $|T| \leq \|S\|$, where $\|S\|$ denotes the total length of strings in S. Breslauer [11] gave an improved algorithm to build the suffix tree of T in $O(|T|)$ time, which is based on Weiner's algorithm.

Another idea for transforming the suffix trie of w into a linear-spaced index structure is to *minimize* it (see Figure 1.1). The resulting structure is the smallest DFA that accepts all suffixes of w [15], which is called the *directed acyclic word graph* (*DAWG*) of w. Blumer et al. [9] introduced DAWGs, and they gave a linear-time algorithm to construct DAWGs, directly and on-line. An improvement of their algorithm to build the DAWG for a given set of strings in linear time was proposed in [10]. DAWGs have been used in several combinatorial algorithms on strings [16, 30, 6, 74].

The duality of suffix trees and DAWGs was pointed out by Chen and Seiferas [12]:

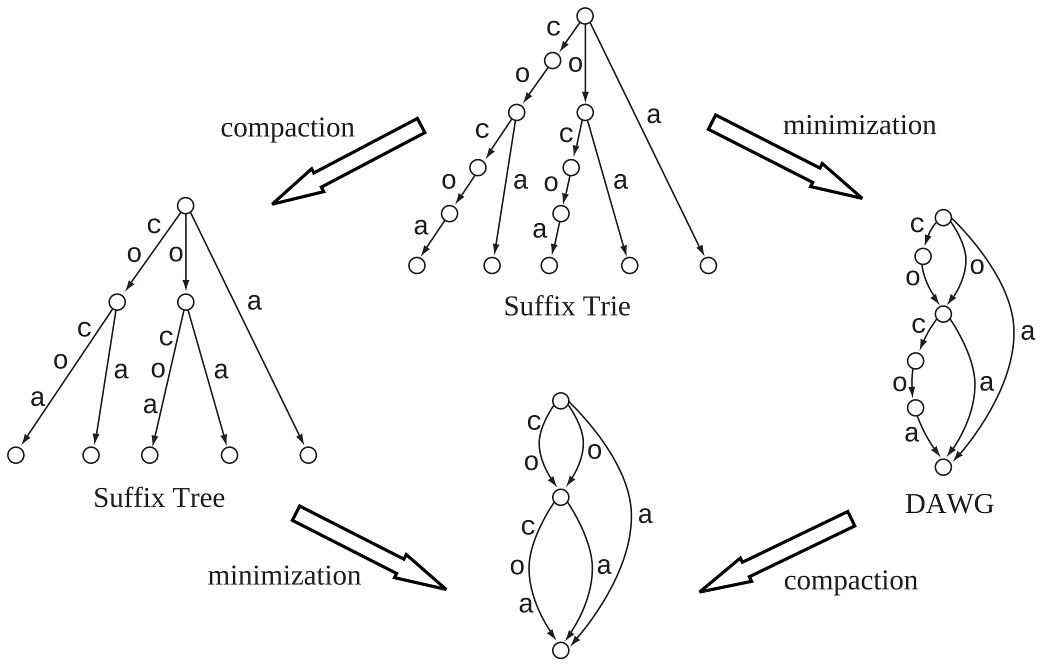


Figure 1.1: Relationship among $STrie(w)$, $STree(w)$, $DAWG(w)$, and $CDAWG(w)$ with $w = \mathtt{cocoa}$.

the DAWG of w and the suffix tree of w^{rev}, where w^{rev} is the reversal of w, can share the same nodes. To be more concrete, the *suffix links* of the DAWG for w can be seen as the edges of the suffix tree for w^{rev}. This implies that, for any input string w, constructing the DAWG for w gives us the suffix tree for w^{rev} as well. To the contrary, it is mentioned in [16] that Weiner's algorithm inherently constructs both the suffix tree of w and the DAWG of w^{rev} for an input string w. A data structure that supports indices of both w and w^{rev} is called a *bidirectional* index structure. One benefit of bidirectional index structures is that after finding p in w, we can easily detect not only the right contexts of p in w (any string y such that py is a substring of w) but also the left contexts of p in w (any string x such that xp is a substring of w).

An interesting fact is that minimizing suffix trees and compacting DAWGs yield the same index structure called *compacted acyclic word graphs* (*CDAWGs*) [10]. As seen in the example of Figure 1.1, CDAWGs have less nodes than both suffix trees and DAWGs. In practice, as well, CDAWGs occupy less memory space than suffix trees and DAWGs [10, 18]. CDAWGs are therefore very attractive, but the first CDAWG-construction algorithm by Blumer et al. [10] was not efficient because its strategy is to construct, for a given string w, the DAWG of w and then convert it to the CDAWG of w. That is, it runs in time linear in the input size, but does not in time linear in the output size. Then, Crochemore and Vérin developed an algorithm to build CDAWGs directly, without constructing DAWGs as intermediate. Their algorithm is based on McCreight's suffix tree construction algorithm.

CDAWGs can also serve as a bidirectional index structure, namely, two CDAWGs for w and w^{rev} can share the same nodes, and they are regarded as one structure called the *symmetric compact directed acyclic word graph* (*SCDAWG*) of w. The SCDAWG of w can be obtained in linear time, in the process of compacting the DAWG of w together with its suffix links corresponding to the suffix tree of w^{rev} [10].

In case that only a limited amount of memory is available, such as when processing an on-line data stream, the *sliding window mechanism* is very useful and often applied actually. Another application of sliding windows is a text compression scheme called *prediction by partial matching* (*PPM*) [14]. Larsson proposed a linear-time algorithm for building and maintaining suffix trees for a sliding window, and the algorithm contributed to reduce space-requirement in PPM* [13]. It is based on Ukkonen's on-line algorithm building suffix trees on-line. Since PPM is known to be the best text-compression scheme from the viewpoint of compression ratio, it is quite meaningful to pursue a reduction of its space requirement.

All the structures mentioned above are automata-oriented. Another form of index structures is arrays. Examples are *suffix arrays* [53], *suffix cacti* [44], *compact suffix arrays* [52], and *compressed suffix arrays* [25, 61]. They are in general more space-economical than those automata-oriented structures (in constant term), but they instead sacrifice searching time. Namely, searching a text w for a pattern p by using an array takes $O(|p| + \log |w|)$ time. Very recently, Abouelhoda et al. [1] introduced *enhanced suffix arrays*, with which searching w for p can be done in $O(|p|)$ time. However, enhanced suffix arrays are regarded as a clever implementation of suffix trees based on arrays, and therefore, it is not very surprising that the search can be done in linear time.

What we have discussed so far is all about the substring pattern matching problem, and in the following we go on to consider more intricate pattern matching problems. We say that p is a *subsequence pattern* of w if p can be obtained by removing zero or more characters from w. The problem of finding a subsequence p in w is also solvable in $O(|p|)$ time by means of an index structure called the *directed acyclic subsequence graph* (*DASG*) of w, invented by Baeza-Yates [5]. The DASG of w is the smallest DFA that accepts all subsequences of w.

The introduction of advanced patterns gives us a good possibility of finding better *rules* and *knowledge* from given data sets. The problem considered is formalized as follows: "Given two sets of strings, find a pattern that is most abundant in one set and rarest in the other." The two sets are often referred to *positive examples* and *negative examples*. Using the machine discovery system BONSAI [63], we can find some knowledge based on substring patterns, which is best in the sense of separating these data sets. A practical algorithm to discover best subsequence patterns was lately given in [28]. It was indeed installed to the BONSAI system and its practical efficiency has been proved experimentally [27].

1.2 Our Contribution

The thesis first focuses on CDAWGs that are known to be a very space-economical index structure for the substring pattern matching. Although a direct construction algorithm for CDAWGs was given by Crochemore and Vérin [18], it was just an off-line algorithm, which leads the following inefficiency: even if we have at the moment the CDAWG of w, to get the CDAWG of wa with any character a we have to reconstruct the CDAWG from scratch. Thus we were not allowed to update a CDAWG with a new character by using any existing algorithms. Our new algorithm for the construction of CDAWGs, however, performs *on-line*, and thus it allows us an easy and efficient update of CDAWGs. The proposed algorithm is based on Ukkonen's suffix tree construction algorithm, together with some essence from Blumer's DAWG-construction algorithm. We show that our algorithm builds for any string w the CDAWG of w in $O(|w|)$ time. We then slightly modify the algorithm so as to be applicable to a set of strings. The improved algorithm is able to construct the CDAWG for a set S of strings in $O(\|S\|)$ time.

A further, deeper analysis of the algorithm above gives us a *generalized algorithm* capable of building, on-line, all of suffix tries, suffix trees, DAWGs, and CDAWGs. All the differences among those are packed in one function of the algorithm, and thus, all we have to do for switching to another mode is to change the function appropriately. In other words, the algorithm clarifies an algorithmic unified view and reveals correspondence between algorithms and definitions of those structures.

We also consider a trie T representing a set S of strings, as an input to the CDAWG construction algorithm. In our trie T the *prefixes* of strings in S are merged together, which means, the more and longer prefixes the strings in S share, the less $|T|$ becomes. Therefore, also in this case, the inequality $|T| \leq \|S\|$ holds. The algorithm we propose here is able to build the CDAWG of T in $O(|T|)$ time. This algorithm is a combination of our on-line algorithm for constructing CDAWGs, and the depth-first search algorithm for T. Moreover, by applying the above-mentioned generalized algorithm to this scheme, we can construct any of the suffix trie, the suffix tree and the DAWG for T as well.

Construction of bidirectional index structures is also discussed in the thesis. We first give an on-line algorithm which constructs, for a given string w, the suffix tree of w and the DAWG of w^{rev} simultaneously, in linear time. Second, we extend this algorithm to linear-time, on-line construction of SCDAWGs. Basically, SCDAWGs have the same range of applications as *affix trees* introduced by Stoye [66, 67], and linear-time algorithm for on-line construction of affix trees was given in [51]. Nevertheless, we have the following fact that, for any string w, the number of nodes in the SCDAWG of w is smaller than in the affix tree of w. Therefore, the efficient construction of SCDAWGs is still meaningful.

Another good feature of affix trees is that they can be updated to both directions, to the right and to the left. The thesis gives its counterpart: an algorithm for bidirectional construction of suffix trees. We also show that the algorithm performs in $O(|w|)$ time for any input string w. Our algorithm turns out to contribute to reducing space-requirement, since it is known that suffix trees generally have fewer nodes than affix trees.

We then focus on the sliding window mechanism for practical string processing with a limited amount of memory space. The thesis indeed gives a space-economical counterpart of Larsson's algorithm, i.e., an algorithm to construct and maintain CDAWGs for a sliding window. This algorithm is based on our on-line algorithm to construct CDAWGs, together with some essence from Larsson's algorithm for suffix trees of a sliding window. Our algorithm runs in $O(|w|)$ time with $O(M)$ space, where M is the width of the sliding window used. Since CDAWGs are more space-economical than suffix trees as is mentioned, our new approach contributes to saving memory space needed in PPM.

The thesis then goes on to consider more complex and difficult string processing. In concrete, given two sets of strings, we consider the problem of finding *episode patterns* [54] that are most common to one set and most uncommon to the other. Episode patterns are a generalized concept of subsequence patterns: a pair $\langle p, k \rangle$ is said to be an episode pattern of w if p is a subsequence of u such that u is a substring of w of length less than or equal to k. We present a practical algorithm to discover the best pair $\langle p, k\rangle$. The ingenious point of our algorithm is that the threshold value k is not given beforehand, that is, it also computes the best value of k for each possible pattern p. Thus, the search space for this problem is $\Sigma^* \times \mathcal{N}$, where Σ is the alphabet used and $\mathcal{N}$ the set of real numbers. What should be emphasized here is that, nevertheless, we can cleverly restrict the search space to the same as that in finding best subsequence p only. Moreover, we employ heuristics for pruning the search tree, inspired by Morishita and Sese [58]. The matching phase of the algorithm is sped up by the use of *episode directed acyclic subsequence graphs* (*EDASGs*) introduced by Troníček [70]. The EDASG of w is an index structure with two kinds of edges, one corresponding to the edges of the DASG of w and the other the DASG of w^{rev}.

Pattern matching with *wildcard* symbols has been studied extensively, for the purpose of more flexible and useful pattern matching. A wildcard $\star$ is called a *variable-length-don't-care* (*VLDC*) symbol which matches any string, and a pattern containing $\star$'s is called a *VLDC pattern*. VLDC patterns are another type of generalization of subsequence patterns; e.g., finding subsequence pattern abc exactly corresponds to finding VLDC pattern $\star a \star b \star c \star$, where $a, b, c \in \Sigma$. Thus, it is highly desired to invent a fast and exact algorithm to discover VLDC patterns best for separating two given sets of strings. A similar pruning heuristics can be employed in this case as well, but some index structure to quickly solve the VLDC pattern matching problem was still needed. Therefor, we present a new index structure called *wildcard directed acyclic word graphs* (*WDAWGs*), with which any VLDC pattern p can be found in $O(|p|)$ time.

A careful observation reveals that the WDAWG of w is inherently composed of *the DAWGs for all suffixes* of w. The collection of all these DAWGs is called the naive *all-suffixes directed acyclic word graph* of w, shortly the naive *ASDAWG* of w, and its minimized version is called the *minimum all-suffixes directed acyclic word graph* (*MAS-DAWG*) of w. The MASDAWG of w can therefore be constructed by applying the DAG-minimization algorithm [60] to the naive ASDAWG of w, but also, we give a non-trivial algorithm to construct the MASDAWG of w *directly*, in time linear in the output size. We also propose another novel algorithm for more space-economical construction of MAS-DAWGs. Then, we introduce the all-suffixes version for CDAWGs, named *minimum all-suffixes compact directed acyclic word graphs* (*MASCDAWGs*). An algorithm which builds the MASCDAWG for a given w is also presented, and we show it runs in linear time with respect to the output size as well.

We then consider *a VLDC pattern within a window* that is composed of a pair $\langle p, k\rangle$, where p is a VLDC pattern and k is a threshold value bounding the length of the occurrence of p in w. This is a generalization of episode patterns. A practical algorithm to discover, from two given sets of strings, the best pair $\langle p, k\rangle$ is also proposed in the thesis. We also execute some computational experiments on genomic sequences, which convince us the actual efficiency of the algorithm.

1.3 Organization of the Thesis

The rest of the thesis is organized as follows.

In Chapter 2 we define some notations, and introduce several properties on strings together with equivalence classes on strings. Chapter 3 is devoted to definitions of several index structures such as suffix tries, suffix trees, DAWGs and CDAWGs, in which we will delve in the following chapters.

In Chapter 4 we present an on-line algorithm that constructs the CDAWG for a given string. We will prove that the proposed algorithm runs in linear time for any string. An extension of this algorithm for construction of the CDAWG for a set of strings is given in Chapter 5, which performs in time linear in the total length of strings in the set. We improve it so as to build the CDAWG for a trie that represents a set of strings. It will be proven that the algorithm runs in time proportional to the number of nodes in the input trie. Chapter 6 presents an algorithm which constructs and maintain CDAWGs under the sliding window mechanism. We show that CDAWGs for a sliding window can be maintained with linear space with respect to the window size given, and in linear time in the length of a given string.

In Chapter 7 we study bidirectional index structures. We give an on-line and linear-time algorithm which, simultaneously, construct the suffix tree of a given string and the DAWG of the reversed string. Moreover, we propose an on-line algorithm for building SCDAWGs, which runs in linear time. An algorithm for bidirectional construction of suffix trees is introduced in Chapter 8. There, we will show its linearity and correctness as well.

In Chapter 9, we give a generic algorithm for construction of the index structures for substring pattern matching. Namely, the algorithm is capable of constructing suffix tries, suffix trees, DAWGs, and CDAWGs, only by changing one function.

Chapter 10 is devoted to introduction of more advanced pattern matching problems such as subsequence pattern matching, episode pattern matching, VLDC pattern matching. We give several efficient algorithms for solving those pattern matching problems.

In Chapter 11 we introduce a new data structure called MASDAWGs, which are essentially the same as WDAWGs with which VLDC pattern matching is quickly solvable. We give a lower-bound of the size of WDAWGs. Another two applications of MASDAWGs can be found in this chapter. Also, we introduce an on-line algorithm for constructing the MASDAWG of a given string, which runs in time linear in the output size. Moreover, in Chapter 12 we present a space-economical algorithm which builds MASDAWGs. Then, we introduce a new data structure called MASCDAWGs, which require less memory space than MASDAWGs. We give an algorithm that constructs MASCDAWGs, and show it runs in time linear in the output size.

In Chapter 13 we give several algorithms for pattern discovery from given string data sets. We first present an efficient algorithm to find the best episode patterns for the purpose of separating two given sets of strings. We secondly give a practical algorithm that exactly finds the best VLDC patterns to distinguish two given sets of strings. Finally, we show some experimental results to evaluate the actual efficiency of our algorithms.

Chapter 2

Preliminaries

2.1 Notation

Let $\mathcal{N}$ be the set of integers. 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*, *substring*, and *suffix* of string $w = xyz$, respectively. The sets of prefixes, substrings, and suffixes of a string w are denoted by $\mathit{Prefix}(w)$, $\mathit{Substr}(w)$, 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 substring of w that begins at position i and ends at position j is denoted by $w[i : j]$ for $1 \leq i \leq j \leq n$. Let $w[i :] = w[i : |w|]$ for $1 \leq i \leq n+1$. For convenience, let $w[i : j] = \varepsilon$ for $j < i$. The reversed string of w, $w[n] \cdots w[2]w[1]$, is denoted by w^{rev}.

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, substrings, and suffixes of the strings in S are denoted by $\mathit{Prefix}(S)$, $\mathit{Substr}(S)$, and $\mathit{Suffix}(S)$, respectively.

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

2.2 Equivalence Relations on Strings

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

Definition 2 Let $w \in \Sigma^*$. The equivalence relations $\equiv^L_w$ and $\equiv^R_w$ on Σ^* are defined by

$$\begin{aligned} x \equiv^L_w y &\Leftrightarrow \mathit{Prefix}(w)x^{-1} = \mathit{Prefix}(w)y^{-1}, \\ x \equiv^R_w y &\Leftrightarrow x^{-1}\mathit{Suffix}(w) = y^{-1}\mathit{Suffix}(w). \end{aligned}$$

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

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

Example 1 Let $w = \mathtt{abcbc}$. For example, $\mathtt{b} \equiv^L_w \mathtt{bc}$ since $\mathit{Prefix}(w)\mathtt{b}^{-1} = \mathit{Prefix}(w)(\mathtt{bc})^{-1} = \{\mathtt{abc}, \mathtt{a}\}$.

All non-degenerate equivalence classes under $\equiv^L_w$ are $[\varepsilon]^L_w = \{\varepsilon\}$, $[\mathtt{b}]^L_w = \{\mathtt{bc}, \mathtt{b}\}$, $[\mathtt{a}]^L_w = \{\mathtt{abcbc}, \mathtt{abcb}, \mathtt{abc}, \mathtt{ab}, \mathtt{a}\}$, $[\mathtt{bcb}]^L_w = \{\mathtt{bcbc}, \mathtt{bcb}\}$, $[\mathtt{c}]^L_w = \{\mathtt{c}\}$, and $[\mathtt{cb}]^L_w = \{\mathtt{cbc}, \mathtt{cb}\}$.

Example 2 Let $w = \mathtt{abcbc}$. For example, $\mathtt{c} \equiv^R_w \mathtt{bc}$ since $\mathtt{c}^{-1}\mathit{Suffix}(w) = (\mathtt{bc})^{-1}\mathit{Suffix}(w) = \{\mathtt{bc}, \varepsilon\}$.

All non-degenerate equivalence classes under $\equiv^R_w$ are $[\varepsilon]^R_w = \{\varepsilon\}$, $[\mathtt{a}]^R_w = \{\mathtt{a}\}$, $[\mathtt{ab}]^R_w = \{\mathtt{ab}\}$, $[\mathtt{abc}]^R_w = \{\mathtt{abc}\}$, $[\mathtt{b}]^R_w = \{\mathtt{b}\}$, $[\mathtt{c}]^R_w = \{\mathtt{bc}, \mathtt{c}\}$, $[\mathtt{cb}]^R_w = \{\mathtt{abcb}, \mathtt{bcb}, \mathtt{cb}\}$, and $[\mathtt{cbc}]^L_w = \{\mathtt{ababc}, \mathtt{bcbc}, \mathtt{cbc}\}$.

Proposition 1 (Blumer et al. [10]) *Let $w \in \Sigma^*$ and $x, y \in \mathit{Substr}(w)$. If $x \equiv^L_w y$, then either x is a prefix of y, or vice versa. If $x \equiv^R_w y$, then either x is a suffix of y, or vice versa.*

Proof. By the definition of $\equiv^L_w$, we have $\mathit{Prefix}(w)x^{-1} = \mathit{Prefix}(w)y^{-1}$. There are three cases to consider:

(1) When $|x| = |y|$. Obviously, $x = y$ in this case. Thus $x \in \mathit{Prefix}(y)$ and $y \in \mathit{Prefix}(x)$.

(2) When $|x| > |y|$. Let u be an arbitrary string in $\mathit{Prefix}(w)$. Assume $u = sx$ with $s \in \Sigma^*$. Then $s \in \mathit{Prefix}(w)x^{-1}$, which results in $s \in \mathit{Prefix}(w)y^{-1}$. Hence, there must exist a string $v \in \mathit{Prefix}(w)$ such that $v = sy$. By the assumption that $|x| > |y|$, we have $|u| > |v|$. From the fact that both u and v are in $\mathit{Prefix}(w)$, it is derived that $v \in \mathit{Prefix}(u)$. Consequently, $y \in \mathit{Prefix}(x)$.

(3) When $|x| < |y|$. By a similar argument to Case (2), we have $x \in \mathit{Prefix}(y)$.

The case of $\equiv^R_w$ is proved similarly. □

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

What $\overset{w}{\overrightarrow{x}}$ (resp. $\overset{w}{\overleftarrow{x}}$) means intuitively is that $\overset{w}{\overrightarrow{x}}$ (resp. $\overset{w}{\overleftarrow{x}}$) is the string obtained by extending x in $[x]^L_w$ (resp. $[x]^R_w$) as long as possible. The following proposition states that each equivalence class in $\equiv^L_w$ ($\equiv^R_w$) other than the degenerate class has a unique longest member.

Proposition 2 (Blumer et al. [10]) *Let $w \in \Sigma^*$. For any string $x \in \mathit{Substr}(w)$, there uniquely exist two strings $\alpha, \beta \in \Sigma^*$ such that $\overset{w}{\overrightarrow{x}} = x\alpha$ and $\overset{w}{\overleftarrow{x}} = \beta x$.*

Proof. Let $\overset{w}{\overrightarrow{x}} = x\alpha$ with $\alpha \in \Sigma^*$. For a contradiction, assume there exists a string $\gamma \in \Sigma^*$ such that $\overset{w}{\overrightarrow{x}} = x\gamma$ and $\gamma \neq \alpha$. By Proposition 1, either $x\alpha \in \mathit{Prefix}(x\gamma)$ or $x\gamma \in \mathit{Prefix}(x\alpha)$ must stand, since $x\alpha \equiv^L_w x\beta$. However, neither of them actually holds since $|\alpha| = |\gamma|$ and $\alpha \neq \gamma$, which yields a contradiction. Hence, α is the only string satisfying $\overset{w}{\overrightarrow{x}} = x\alpha$. The case of $\overset{w}{\overleftarrow{x}} = \beta x$ can be proven similarly. □

Proposition 3 *Let $w \in \Sigma^*$ and $x \in \mathit{Substr}(w)$. Assume $\overset{w}{\overrightarrow{x}} = x$. Then, for any $y \in \mathit{Suffix}(x)$, $\overset{w}{\overrightarrow{y}} = y$.*

Proof. Assume contrarily that there uniquely exists a string $\alpha \in \Sigma^+$ such that $\overset{w}{\overrightarrow{y}} = y\alpha$. Since $y \in \mathit{Suffix}(x)$, x is always followed by α in w. It implies that $\mathit{Prefix}(w)x^{-1} = \mathit{Prefix}(w)(x\alpha)^{-1}$, and therefore we have $x \equiv^L_w x\alpha$. Since $|\alpha| > 0$, $\overset{w}{\overrightarrow{x}}$ is not the longest in $[x]^L_w$, which is a contradiction. Hence, $\overset{w}{\overrightarrow{y}} = y$. □

Analogously, we have:

Proposition 4 *Let $w \in \Sigma^*$ and $x \in Substr(w)$. Assume $\overset{w}{\overleftarrow{x}} = x$. Then, for any $y \in Prefix(x)$, $\overset{w}{\overleftarrow{y}} = y$.*

Proof. By Proposition 3. □

Proposition 5 *Let $w \in \Sigma^*$. For any string $x \in Suffix(w)$, $\overset{w}{\overrightarrow{x}} = x$. Symmetrically, for any string $y \in Prefix(w)$, $\overset{w}{\overleftarrow{y}} = y$.*

Proof. By Proposition 2 there uniquely exists a string $\alpha \in \Sigma^*$ such that $\overset{w}{\overrightarrow{x}} = x\alpha$. Since $x \in Suffix(w)$, $\alpha = \varepsilon$. It is similarly proven that, letting $\overset{w}{\overleftarrow{y}} = \beta y$ with $\beta \in \Sigma^*$, we have $\beta = \varepsilon$. □

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

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

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

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

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

1. $x, y \in Substr(w)$ and $\overset{w}{\overleftrightarrow{x}} = \overset{w}{\overleftrightarrow{y}}$, or
2. $x \notin Substr(w)$ and $y \notin Substr(w)$.

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

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

Example 3 Let $w = \texttt{abcbc}$. For example, $\texttt{ab} \equiv_w \texttt{bcb}$ since $\overset{w}{\overleftrightarrow{\texttt{ab}}} = \overset{w}{\overleftrightarrow{\texttt{bcb}}} = \texttt{abcbc}$.

All non-degenerate equivalence classes under $\equiv_w$ are $[\varepsilon]_w = \{\varepsilon\}$, $[\texttt{b}]_w = \{\texttt{bc}, \texttt{b}, \texttt{c}\}$, and $[\texttt{a}]_w = \{\texttt{abcbc}, \texttt{abcb}, \texttt{abc}, \texttt{ab}, \texttt{a}, \texttt{bcbc}, \texttt{bcb}\,\texttt{cbc}, \texttt{cb}\}$.

Lemma 1 (Blumer et al. [10]) *The equivalence relation $\equiv_w$ is the transitive closure of the relation $\equiv^L_w \cup \equiv^R_w$.*

It follows from the lemma above that

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

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

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

Lemma 2 (Blumer et al. [9, 10]) *Assume that $|w| > 1$. The number of the non-degenerate equivalence classes in $\equiv^L_w$ (or $\equiv^R_w$) is at most $2|w| - 1$. The number of the non-degenerate equivalence classes in $\equiv_w$ is at most $|w| + 1$.*

2.3 Graphs and Trees

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

In a directed graph $G = (V, E)$, 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 $(1 \le i \le n)$. The *depth* of the path is n. A path with $u_0 = u_n$ is called a *cycle*. If G has no cycles, 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 zero 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) is an edge of 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.

2.3.1 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 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 $\mathit{Trie}(S)$. In some cases it is favorable that all strings in S are associated with leaf nodes of $\mathit{Trie}(S)$. In such case, we consider the set S' such that

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

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

2.4 Deterministic Finite Automata

A *deterministic finite automaton* (*DFA* for short) is a quintuplet $M = (Q, \Sigma, \delta, q_0, F)$ with the following components:

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.

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

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.

Chapter 3

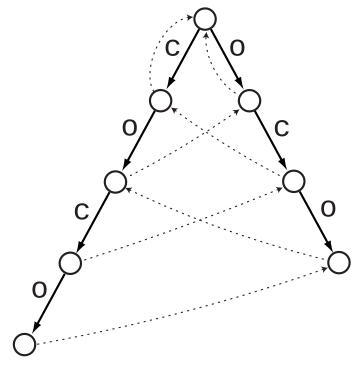
Data Structures for Substring Pattern Matching

The *substring matching problem* is the most fundamental and important problem in string processing. The problem is defined as follows:

Definition 7 (Substring Pattern Matching Problem)
Instance: text $w \in \Sigma^*$ and pattern $p \in \Sigma^*$.
Determine: whether p is a substring of w.

The algorithm by Knuth, Morris and Pratt [45] is capable of solving the above problem in $O(|w|)$ time using $O(|p|)$ space. Namely, their algorithm constructs a DFA that accepts any string which contains p as a substring, and runs the DFA over a given text w. Therefore, it is very useful and fast when p is fixed and w is flexible. In the opposite case where p is flexible and w is fixed, however, their algorithm is no longer adequate since whenever we get a new pattern p, we have to reconstruct the DFA and run it over the same string w which is often tremendously long. This implies that we should consider the use of some data structure that supports indices of w.

In this chapter of the thesis, we give definitions of data structures such as *suffix tries*, *suffix trees*, *directed acyclic word graphs* (*DAWGs*), and *compact directed acyclic word graphs* (*CDAWGs*). The suffix trie, the suffix tree, the DAWG, and the CDAWG of string $w \in \Sigma^*$ are denoted by $STrie(w)$, $STree(w)$, $DAWG(w)$, and $CDAWG(w)$, respectively. The language recognized by any of these data structures is $Substr(w)$, and therefore, they can serve as *index structures* with which the substring pattern matching problem is solvable in $O(|p|)$ time for any given pattern p.



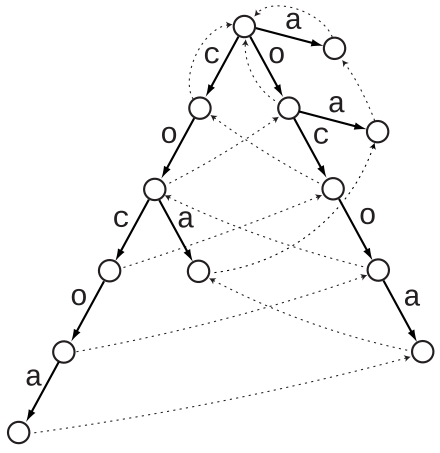


Figure 3.1: *STrie*(`coco`) on the left, and *STrie*(`cocoa`) on the right. The solid arrows represent the edges, and the dotted arrows denote the suffix links.

We define them as edge-labeled graphs (V, E) with $E \subseteq V \times \Sigma^+ \times V$. 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 [55, 73, 9, 10, 18].

3.1 Suffix Tries

Definition 8 *STrie*(w) is the tree (V, E) such that

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

and its suffix links are the set

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

Note that *STrie*(w) is a DFA that accepts all strings in *Substr*(w). Moreover, observe that *STrie*(w) = *Trie*(*Suffix*(w)). The root node of *STrie*(w) corresponds to ε. When *Suffix*$(w) - \{\varepsilon\}$ has the prefix property, every string in *Suffix*$(w) - \{\varepsilon\}$ is represented by a leaf node in *STrie*(w). *STrie*(`coco`) and *STrie*(`cocoa`) are displayed in Figure 3.1 together with their suffix links.

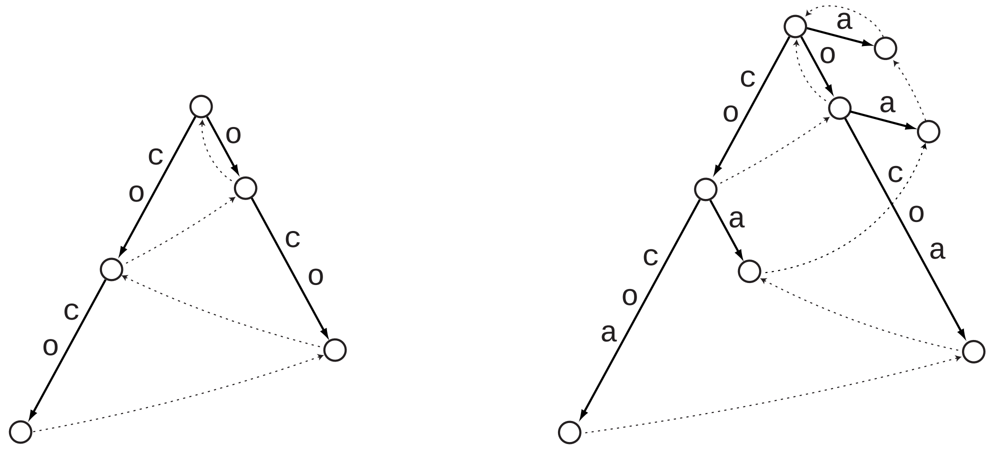


Figure 3.2: *STree*(coco) on the left, and *STree*(cocoa) on the right. The solid arrows represent the edges, and the dotted arrows denote the suffix links.

3.2 Suffix Trees

Definition 9 *STree*(w) is the tree (V, E) such that

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

and its suffix links are the set

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

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

Remark that the node set of *STree*(w) is a subset of that of *STrie*(w), as seen in the definitions. It means that a string in *Substr*(w) might be represented on an edge in *STree*(w). In this case, we say that the string is represented on an *implicit* node. Conversely, every string in the node set V of *STree*(w) is said to be represented in an *explicit* node. For example, in *STree*(coco) of Figure 3.2, string c is represented on an implicit node, while string co is on an explicit node. *STree*(w) can be seen as the compacted version of *STrie*(w) with "$\overset{w}{\overrightarrow{(\cdot)}}$ operation". Compare *STrie*(cocoa) in Figure 3.1 and *STree*(cocoa) in Figure 3.2. *STree*(cocoa) can be obtained by removing any internal nodes of out-degree one from *STrie*(cocoa), and the suffix links associated with the removed nodes are also deleted. However, this approach cannot derive $STree(\texttt{coco})$ from $STrie(\texttt{coco})$ (see Figure 3.1 and Figure 3.2). That is, even if a node $\overset{w}{\overrightarrow{x}}$ is of out-degree one in $STrie(w)$, it is not removed if $\overset{w}{\overrightarrow{x}} \in Suffix(w)$.

The following theorem follows from Definition 9 and Lemma 2.

Theorem 1 (McCreight [55]) *Let* $STree(w) = (V, E)$. *Assume* $|w| > 1$. *Then* $|V| \leq 2|w| - 1$ *and* $|E| \leq 2|w| - 2$.

3.3 DAWGs

Definition 10 $DAWG(w)$ is the directed acyclic graph (V, E) such that

$$\begin{aligned} V &= \{[x]^R_w \mid x \in Substr(w)\}, \\ E &= \{([x]^R_w, a, [xa]^R_w) \mid x, xa \in Substr(w) \text{ and } a \in \Sigma\}, \end{aligned}$$

and its suffix links are the set

$$F = \{([ax]^R_w, [x]^R_w) \mid x, ax \in Substr(w),\ a \in \Sigma, \text{ and } [ax]^R_w \neq [x]^R_w\}.$$

The node $[\varepsilon]^R_w = \{\varepsilon\}$ is called the *source* node of $DAWG(w)$. For any $w \in \Sigma^*$ there uniquely exists a node of out-degree zero in $DAWG(w)$. This node is called the *sink* node of $DAWG(w)$, and corresponds to $[w]^R_w$.

Definition 11 Let $[x]^R_w$ be a node of $DAWG(w)$. The *length* of the node is the length of the representative of $[x]^R_w$, namely, $|\overset{w}{\overleftarrow{x}}|$.

The length of node $[x]^R_w$ is denoted by $length([x]^R_w)$.

Definition 12 Assume $([x]^R_w, a, [y]^R_w)$ is an edge of $DAWG(w)$, where $a \in \Sigma$, $w \in \Sigma^*$, and $x, y \in Substr(w)$. If $length([y]^R_w) = length([x]^R_w) + |a| = length([x]^R_w) + 1$, the edge is said to be *solid*. Otherwise, it is said to be *non-solid*.

For example, in $DAWG(w)$ of Figure 3.3 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 $DAWG(w)$ is a non-degenerate equivalence class with respect to $\equiv^R_w$. See $STrie(\texttt{cocoa})$ in Figure 3.1. Observe that, by 'merging' isomorphic subtrees of $STrie(w)$ according to the equivalence classes under $\equiv^R_w$, $DAWG(\texttt{cocoa})$ is obtained. In this sense, $DAWG(w)$ can be seen as the minimized version of $STrie(w)$ with "$[(\cdot)]^R_w$ operation". In fact, we have the following theorem.

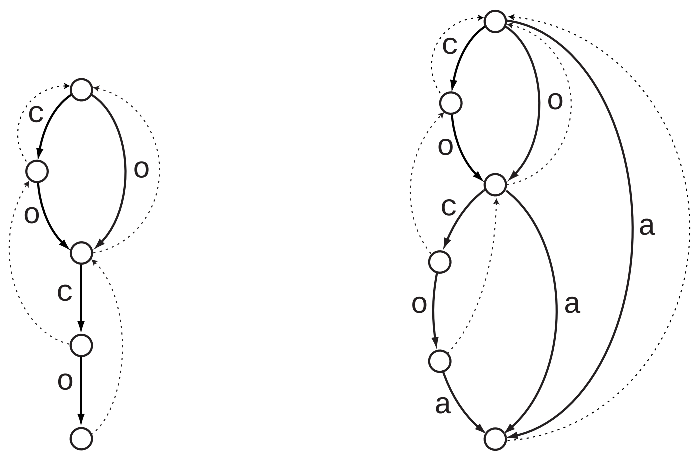


Figure 3.3: *DAWG*(`coco`) on the left, and *DAWG*(`cocoa`) on the right. The solid arrows represent the edges, and the dotted arrows denote the suffix links.

Theorem 2 (Crochemore [15]) *For any $w \in \Sigma^*$, $DAWG(w)$ is the smallest DFA that recognizes all suffixes of w.*

Suppose that ax is the shortest member of $[ax]^R_w$, for some character $a \in \Sigma$ and string $x \in Substr(w)$. Then the suffix link of node $[ax]^R_w$ of $DAWG(w)$ points to the node $[x]^R_w$ (for example, see nodes $[\mathtt{oco}]^R_w$ and $[\mathtt{co}]^R_w$ of $DAWG(\mathtt{cocoa})$ in Figure 3.3).

The following theorem follows from Definition 10 and Lemma 2.

Theorem 3 (Blumer et al. [9]) *Let $DAWG(w) = (V, E)$. Assume $|w| > 1$. Then $|V| \leq 2|w| - 1$ and $|E| \leq 3|w| - 3$.*

3.4 CDAWGs

Definition 13 *CDAWG*(w) is the directed acyclic graph (V, E) such that

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

and its suffix links are the set

$$F = \{([\overset{w}{\overrightarrow{ax}}]^R_w, [\overset{w}{\overrightarrow{x}}]^R_w) \mid x, ax \in Substr(w),\ a \in \Sigma,\ \overset{w}{\overrightarrow{ax}} = a \cdot \overset{w}{\overrightarrow{x}}, \text{ and } [\overset{w}{\overrightarrow{x}}]^R_w \neq [\overset{w}{\overrightarrow{ax}}]^R_w\}.$$

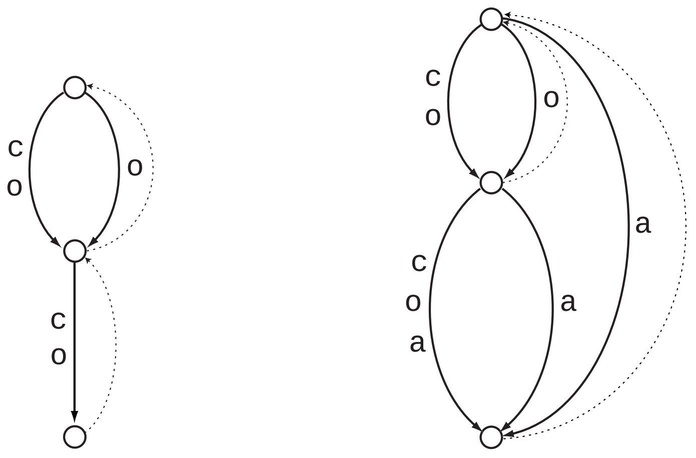


Figure 3.4: *CDAWG*(`coco`) on the left, and *CDAWG*(`cocoa`) on the right. The solid arrows represent the edges, and the dotted arrows denote the suffix links.

The node $[\overset{w}{\overrightarrow{\varepsilon}}]^R_w = \varepsilon$ is called the *source* node, and the node $[\overset{w}{\overrightarrow{w}}]^R_w$, which is of out-degree zero, is called the *sink* node of *CDAWG*(w). Notice there is a one-to-one correspondence between $[\overset{w}{\overrightarrow{x}}]^R_w$ and $\overset{w}{\overleftrightarrow{x}}$ according to Corollary 1.

Definition 14 Let $[\overset{w}{\overrightarrow{x}}]^R_w$ be a node of *CDAWG*(w). The *length* of the node is $\left|\overset{w}{\overleftarrow{(\overset{w}{\overrightarrow{x}})}}\right| = |\overset{w}{\overleftrightarrow{x}}|$.

The length of node $[\overset{w}{\overrightarrow{x}}]^R_w$ is denoted by *length*$([\overset{w}{\overrightarrow{x}}]^R_w)$.

Definition 15 Assume $([\overset{w}{\overrightarrow{x}}]^R_w, \alpha, [\overset{w}{\overrightarrow{y}}]^R_w)$ is an edge of *CDAWG*(w), where $\alpha, w \in \Sigma^*$, and $x, y \in$ *Substr*(w). If *length*$([\overset{w}{\overrightarrow{y}}]^R_w)$ = *length*$([\overset{w}{\overrightarrow{x}}]^R_w) + |\alpha|$, the edge is said to be *solid*. Otherwise, it is said to be *non-solid*.

In *CDAWG*(w) of Figure 3.4, where $w =$ `coco`, edge $([\overset{w}{\overrightarrow{\varepsilon}}]^R_w, \texttt{co}, [\overset{w}{\overrightarrow{\texttt{co}}}]^R_w)$ is solid, while edge $([\overset{w}{\overrightarrow{\varepsilon}}]^R_w, \texttt{o}, [\overset{w}{\overrightarrow{\texttt{co}}}]^R_w)$ is non-solid.

It follows from the definitions that *CDAWG*(w) is the minimization of *STree*(w) with "$[(\cdot)]^R_w$ operation". In fact, *CDAWG*(`cocoa`) in Figure 3.4 can be obtained by 'merging' the isomorphic subtrees of *STree*(`cocoa`) according to the equivalence classes under $\equiv^R_w$. Similarly, *CDAWG*(w) can also be seen as the compaction of *DAWG*(w) with "$\overset{w}{\overrightarrow{(\cdot)}}$ operation". Indeed *CDAWG*(`cocoa`) is obtained by compacting edges of *DAWG*(`cocoa`) in Figure 3.3 due to the equivalence classes under $\equiv^L_w$.

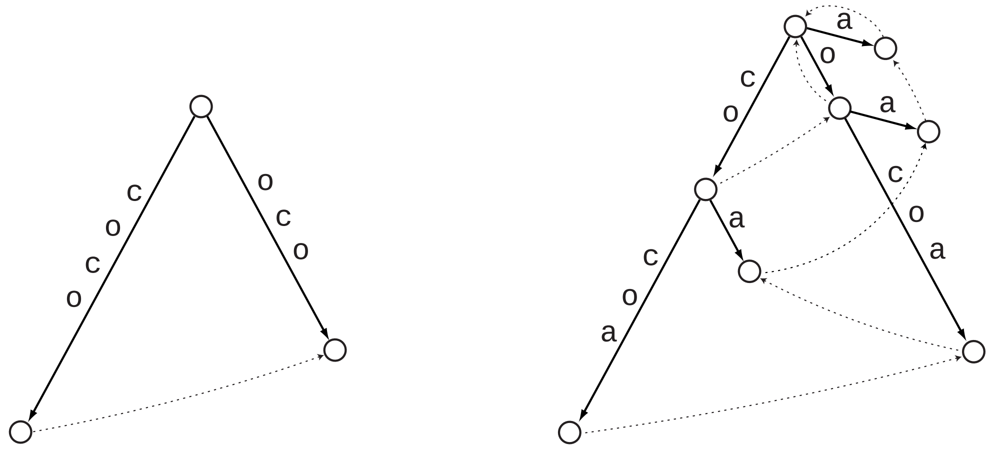


Figure 3.5: *STree*′(`coco`) on the left, and *STree*′(`cocoa`) on the right. The solid arrows represent the edges, and the dotted arrows denote the suffix links.

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

The following theorem follows from Definition 13 and Lemma 2.

Theorem 4 (Blumer et al. [10], Crochemore and Vérin [18])
Let $CDAWG(w) = (V, E)$. *Assume* $|w| > 1$. *Then* $|V| \leq |w| + 1$ *and* $|E| \leq 2|w| - 2$.

3.5 Suffix Trees Redefined

For any string $w \in \Sigma^*$, let *STree*′(w) denote the tree obtained by removing all internal nodes of out-degree one from *STree*(w). As seen in Figure 3.5, nodes $\overset{w}{\overrightarrow{\texttt{co}}}$ and $\overset{w}{\overrightarrow{\texttt{o}}}$ in *STree*(`coco`) are omitted in *STree*′(`coco`) together with their suffix links. Ukkonen's suffix tree construction algorithm [73] builds *STree*′(w), not *STree*(w). The following preparation is necessary for a formal definition of *STree*′(w).

We introduce a relation X_w over Σ^* such that

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

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

Proposition 6 *For any string $w \in \Sigma^*$, $\equiv^L_w$ is a refinement of $\equiv'^L_w$.*

Proof. Let x, y be any strings in $\mathit{Substr}(w)$ and assume $x \equiv^L_w y$. According to Proposition 1, we firstly assume that $x \in \mathit{Prefix}(y)$. It follows from Proposition 2 that there uniquely exist strings $\alpha, \beta \in \Sigma^*$ such that $\overset{w}{\overrightarrow{x}} = x\alpha$ and $\overset{w}{\overrightarrow{y}} = y\beta$. Note that $\beta \in \mathit{Suffix}(\alpha)$. Let $\gamma \in \Sigma^*$ be the string satisfying $\alpha = \gamma\beta$. Then γ is the sole string such that $x\gamma = y$. By the definition of $\equiv'^L_w$, we have $x \equiv'^L_w y$. A similar argument holds in case that $y \in \mathit{Prefix}(x)$. □

Corollary 2 *For any $w \in \Sigma^*$, every equivalence class under $\equiv'^L_w$ is a union of one or more equivalence classes under $\equiv^L_w$.*

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

Example 4 Let $w = \texttt{abcbc}$. All equivalence classes in $\equiv'^L_w$ are $[\varepsilon]'^L_w = \{\varepsilon\}$, $[\texttt{a}]'^L_w = \{\texttt{abcbc}, \texttt{abcb}, \texttt{abc}, \texttt{ab}, \texttt{a}\}$, $[\texttt{b}]'^L_w = \{\texttt{bcbc}, \texttt{bcb}, \texttt{bc}, \texttt{b}\}$, and $[\texttt{c}]'^L_w = \{\texttt{cbc}, \texttt{cb}, \texttt{c}\}$.

Note the differences between the above example and Example 1 with respect to $\equiv^L_w$.

The longest member of $[x]'^L_w$ is denoted by $\overset{w}{\overset{\Longrightarrow}{x}}$. We are now ready to define $\mathit{STree}'(w)$.

Definition 16 *$\mathit{STree}'(w)$ is the tree (V, E) such that*

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

and its suffix links are the set

$$F = \{(\overset{w}{\overset{\Longrightarrow}{ax}}, \overset{w}{\overset{\Longrightarrow}{x}}) \mid x, xa \in \mathit{Substr}(w),\ a \in \Sigma,\ \overset{w}{\overset{\Longrightarrow}{ax}} = a \cdot \overset{w}{\overset{\Longrightarrow}{x}}\}.$$

Remark that $\mathit{STree}'(w)$ can be obtained by replacing $\overset{w}{\overrightarrow{(\cdot)}}$ in $\mathit{STree}(w)$ with $\overset{w}{\overset{\Longrightarrow}{(\cdot)}}$.

From here on, we explore some relationships between $\overset{w}{\overrightarrow{(\cdot)}}$ and $\overset{w}{\overset{\Longrightarrow}{(\cdot)}}$.

Lemma 3 *Let $w \in \Sigma^*$. For any $x \in \mathit{Substr}(w)$, $\overset{w}{\overrightarrow{x}}$ is a prefix of $\overset{w}{\overset{\Longrightarrow}{x}}$. If $\overset{w}{\overrightarrow{x}} \neq \overset{w}{\overset{\Longrightarrow}{x}}$, then $\overset{w}{\overrightarrow{x}} \in \mathit{Suffix}(w)$.*

Proof. We can prove that $\overset{w}{\overrightarrow{x}} \in \mathit{Prefix}(\overset{w}{\overset{\Longrightarrow}{x}})$ by Proposition 1 and Corollary 2. Now suppose $\overset{w}{\overrightarrow{x}} \neq \overset{w}{\overset{\Longrightarrow}{x}}$. Let $\overset{w}{\overrightarrow{x}} = x\beta$ with $\beta \in \Sigma^+$. Supposing $\overset{w}{\overset{\Longrightarrow}{x}} = x\alpha$ with $\alpha \in \Sigma^+$, we have

$\beta \in \textit{Prefix}(\alpha)$. Let $\beta\gamma = \alpha$ with $\gamma \in \Sigma^*$. By the assumption $\overset{w}{\overrightarrow{x}} \neq \overset{w}{\overset{\Longrightarrow}{x}}$, we have $x\beta \not\equiv^L_w x\alpha$, although γ is the sole string that follows $\overset{w}{\overrightarrow{x}}$ in w since $\overset{w}{\overset{\Longrightarrow}{x}} = x\alpha = x\beta\gamma = \overset{w}{\overrightarrow{x}} \cdot \gamma$. This means that x is a suffix of w followed by *no* character. $\square$

See Example 1 and Example 4 to confirm the above lemma.

Lemma 4 *Let $w \in \Sigma^*$ and $x \in \textit{Suffix}(w)$. If $x \notin \textit{Prefix}(y)$ for any string $y \in \textit{Substr}(w) - \{x\}$, then $\overset{w}{\overrightarrow{x}} = \overset{w}{\overset{\Longrightarrow}{x}}$.*

Proof. The precondition implies that there is no character $a \in \Sigma$ satisfying $xa \in \textit{Substr}(w)$. Thus we have $\overset{w}{\overset{\Longrightarrow}{x}} = x$. On the other hand, we obtain $\overset{w}{\overrightarrow{x}} = x$ by Proposition 5, because $x \in \textit{Suffix}(w)$. Hence $\overset{w}{\overrightarrow{x}} = \overset{w}{\overset{\Longrightarrow}{x}}$. $\square$

The following corollary gives a sufficient condition so that $\textit{STree}'(w) = \textit{STree}'(w)$.

Corollary 3 *Let $w \in \Sigma^*$. If $\textit{Suffix}(w) - \{\varepsilon\}$ has the prefix property, $\textit{STree}'(w) = \textit{STree}(w)$.*

Indeed, $\textit{STree}(\texttt{cocoa})$ in Figure 3.2 and $\textit{STree}'(\texttt{cocoa})$ in Figure 3.5 are the same, where $\textit{Suffix}(\texttt{cocoa})$ has the prefix property. According to the above corollary, using an end-marker \$ that occurs nowhere in w, we have $\textit{STree}(w\$) = \textit{STree}'(w\$)$ for any $w \in \Sigma^*$.

3.6 CDAWGs Redefined

Similar to $\textit{STree}'(w)$, for any string $w \in \Sigma^*$, $\textit{CDAWG}'(w)$ has no internal node of out-degree one.

Definition 17 $\textit{CDAWG}'(w)$ is the directed acyclic graph (V, E) such that

$$\begin{aligned} V &= \{[\overset{w}{\overset{\Longrightarrow}{x}}]^R_w \mid x \in \textit{Substr}(w)\}, \\ E &= \{([\overset{w}{\overset{\Longrightarrow}{x}}]^R_w, a\beta, [\overset{w}{\overset{\Longrightarrow}{xa}}]^R_w) \mid x, xa \in \textit{Substr}(w),\ a \in \Sigma,\ \beta \in \Sigma^*,\ \overset{w}{\overset{\Longrightarrow}{xa}} = xa\beta, \text{ and } \overset{w}{\overrightarrow{x}} \neq \overset{w}{\overset{\Longrightarrow}{xa}}\}, \end{aligned}$$

and its suffix links are the set

$$F = \{([\overset{w}{\overset{\Longrightarrow}{ax}}]^R_w, [\overset{w}{\overset{\Longrightarrow}{x}}]^R_w) \mid x, ax \in \textit{Substr}(w),\ a \in \Sigma,\ \overset{w}{\overset{\Longrightarrow}{ax}} = a \cdot \overset{w}{\overset{\Longrightarrow}{x}}, \text{ and } [\overset{w}{\overset{\Longrightarrow}{x}}]^R_w \neq [\overset{w}{\overset{\Longrightarrow}{ax}}]^R_w\}.$$

As in case of $\textit{STree}'(w)$ and $\textit{STree}(w)$, the above definition equals the one obtained by substituting "$\overset{w}{\overrightarrow{(\cdot)}}$ operation" with "$\overset{w}{\overset{\Longrightarrow}{(\cdot)}}$ operation" in Definition 13 for $\textit{CDAWG}'(w)$.

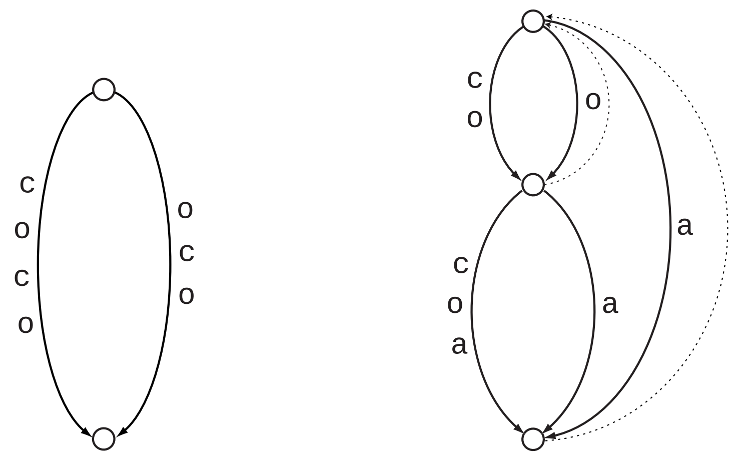


Figure 3.6: *CDAWG′*(coco) on the left, and *CDAWG′*(cocoa) on the right. The solid arrows represent the edges, and the dotted arrows denote the suffix links.

Corollary 4 *Let $w \in \Sigma^*$. If Suffix$(w) - \{\varepsilon\}$ has the prefix property, CDAWG′(w) = CDAWG(w).*

Remark that *CDAWG′*(cocoa) in Figure 3.6 and *CDAWG*(cocoa) in Figure 3.4 are the same, where *Suffix*(cocoa) has the prefix property. Again, it derives from the above corollary that, using an end-marker \$, we have *CDAWG′*$(w\$)$ = *CDAWG*$(w\$)$ for any $w \in \Sigma^*$.

Chapter 4

On-Line Construction of CDAWGs

In this chapter we focus our attention on compact directed acyclic word graphs (CDAWGs) introduced by Blumer et al. [10]. Crochemore and Vérin showed a relationship among suffix tries, suffix trees, DAWGs, and CDAWGs [18]. Suffix trees (resp. DAWGs) are the compacted (resp. minimized) version of suffix tries, as shown in Figure 1.1. Similarly, CDAWGs can be obtained by either compacting DAWGs or minimizing suffix trees. This implies that CDAWGs have less nodes than the other three index structures.

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 [10, 18]. 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 $CDAWG(w)$ for a given string w was presented in [9]. It once builds $DAWG(w)$, then removes every node of out-degree one and modifies its edges accordingly, so that the resulting structure becomes $CDAWG(w)$. 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 [18]: a linear-time algorithm to construct CDAWGs *directly*. Their algorithm is based on McCreight's suffix tree construction algorithm [55]. Both algorithms are *off-line*, that is, the whole input string has to be known beforehand. Thus, the structure (suffix tree or CDAWG) has to be rebuilt from scratch, when a new character is added to the input string. Table 4.1 summarizes some properties of typical algorithms to construct index structures. As seen there, a missing piece was an *on-line algorithm for constructing CDAWGs*.

Index Structure	*Algorithm*	*linear time*	*on-line*	*set of strings*
suffix tries	Ukkonen [73]	—	$\surd$	$\surd$
suffix trees	Weiner [77]	$\surd$		
	McCreight [55]	$\surd$		
	Ukkonen [73]	$\surd$	$\surd$	$\surd$
DAWGs	Blumer et al. [9]	$\surd$	$\surd$	
	Blumer et al. [10]	$\surd$	$\surd$	$\surd$
CDAWGs	Blumer et al. [10]	$\surd$		$\surd$
	Crochemore and Vérin [18]	$\surd$		
	Inenaga et al. [39]	$\surd$	$\surd$	$\surd$

Table 4.1: The properties of algorithms for construction of index structures.

In this chapter, we present a new linear-time algorithm which, for a given string w, directly constructs $\mathit{CDAWG}'(w)$. It is based on Ukkonen's algorithm that constructs $\mathit{STree}'(w)$ in linear time [73]. 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 [18], 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 [10] 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 rebuilt from scratch. Instead, our algorithm permits us the addition of a new string to the set.

This result primarily appeared in [39].

4.1 On-Line Construction of Suffix Tries

The on-line CDAWG construction algorithm we will give later on is based on Ukkonen's on-line suffix tree construction algorithm [73]. Moreover, Ukkonen's algorithm is based on an intuitive on-line algorithm that constructs suffix tries.

For a string $x \in \mathit{Substr}(w)$, let $\mathit{suf}(x)$ denote the node reachable via the suffix link of the node x. It derives from Definition 8 that $\mathit{suf}(x) = y$ for some $y \in \mathit{Substr}(w)$ 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. We assume the $\perp$ node corresponds to the *eliminator* ξ defined below.

Definition 18 For any $a \in \Sigma$, we define the *eliminator* ξ by

$$a\xi = \xi a = \varepsilon$$

and $|\xi| = -1$. We define that $\xi \in \mathit{Prefix}(\varepsilon)$ and $\xi \in \mathit{Suffix}(\varepsilon)$, but $\xi \notin \mathit{Prefix}(x)$ and $\xi \notin \mathit{Suffix}(x)$ for any $x \in \Sigma^+$.

The edge $(\perp, \Sigma, \varepsilon)$ is now consistently defined as well as other edges, since $\xi a = \varepsilon$ for any $a \in \Sigma$. 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 19 Let $a \in \Sigma$ and $u \in \Sigma^*$. The *longest repeated suffix* (*LRS*) of ua is the longest element of set $\mathit{Substr}(u) \cap \mathit{Suffix}(ua)$.

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

The suffixes of $w[1:i+1]$ can be divided into the following two groups, by $\mathit{LRS}(w[1:i+1])$.

(1) Suffixes $w[h:i+1]$ for $1 \leq h \leq j$ where $\mathit{LRS}(w[j+1:i+1]) = w[1:i+1]$.

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

The group (2) is empty in case that $\mathit{LRS}(w[1:i+1]) = \varepsilon$, 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 $\mathit{LRS}(w[h:i+1])$ 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 $\mathit{LRS}(w[j+1:i])$, the algorithm stops and the update is then completed. The node with respect to $\mathit{LRS}(w[j+1:i+1])$ is called the *end point* of $\mathit{STrie}(w[1:i+1])$.

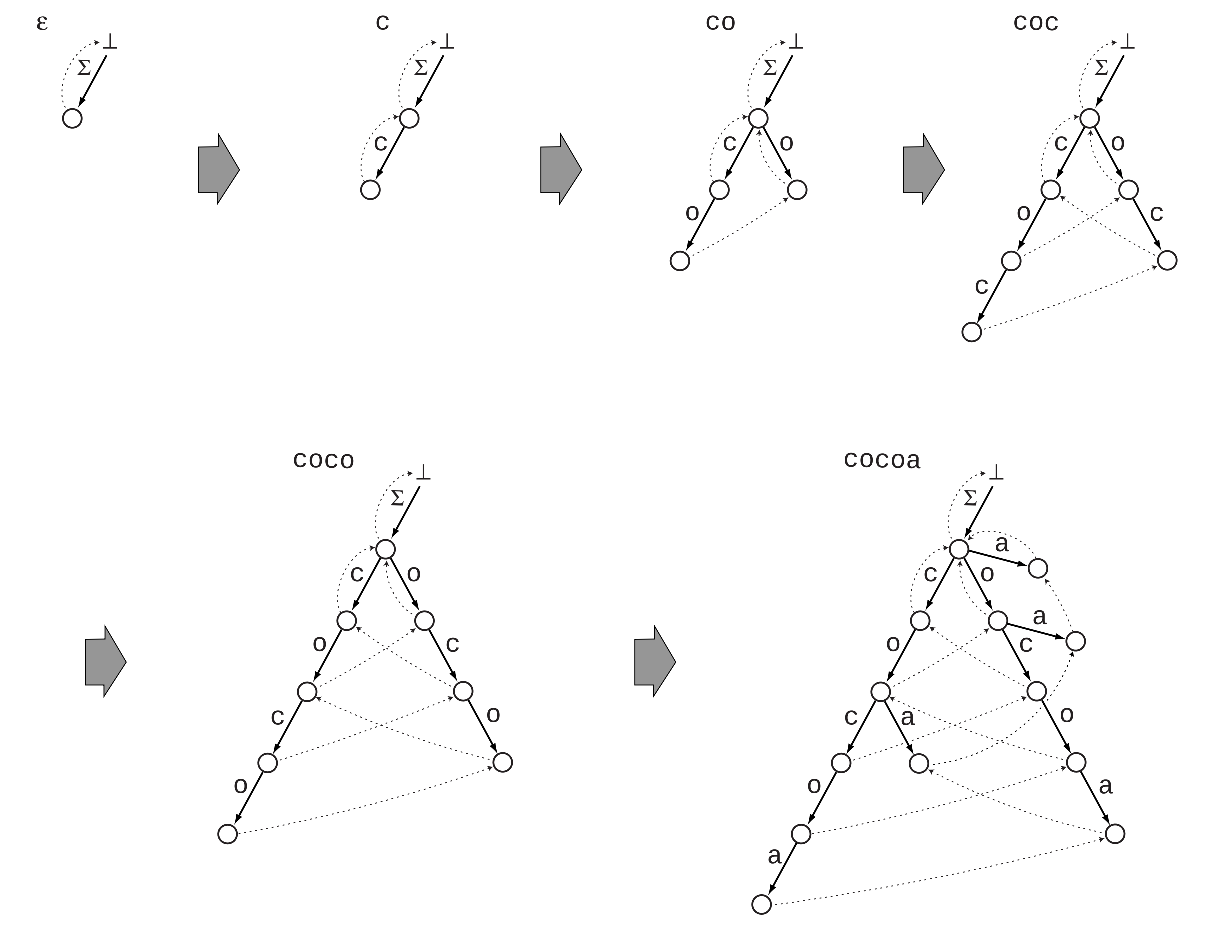


Figure 4.1: On-line construction of $STrie(w)$ with $w =$ `cocoa`.

The on-line construction of $STrie($`cocoa`$)$ is shown in Figure 4.1.

Unfortunately, $STrie(w)$ cannot be constructed in $O(|w|)$ time, since it requires $O(|w|^2)$ space. Still, we have the following theorem.

Theorem 5 (Ukkonen [73]) *Assume Σ is a fixed alphabet. For any string $w \in \Sigma^*$, $STrie(w)$ can be constructed on-line and in linear time and space with respect to the output size.*

4.2 On-Line Construction of Suffix Trees

In this section we firstly summarize Ukkonen's suffix tree construction algorithm in the comparison with the previous suffix trie algorithm. Figure 4.2 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 own 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.

4.2.1 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 4.1. As stated in Definition 16, an edge of *STree*$'(w)$ is labeled by a string $\alpha \in \mathit{Substr}(w)$. 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 $y \in \mathit{Substr}(w)$, where $\overrightarrow{y}^{w} \neq y$, can be represented by a pair $(\overrightarrow{x}^{w}, \alpha)$ of an explicit node $\overrightarrow{x}^{w}$ and a string $\alpha \in \mathit{Substr}(w)$ 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 more than one reference pair for 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 $1 \leq i \leq |w|$. 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 $1 \leq l \leq j'$ where $w[j'+1:i]$ is the LRS of $w[1:i]$.

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

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, $\overset{\overset{w[1:i]}{\Longrightarrow}}{w[l:i]} = w[l:i]$ and $\overset{\overset{w[1:i+1]}{\Longrightarrow}}{w[l:i+1]} = w[l:i+1]$. 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 $(\overset{\overset{w[1:i]}{\Longrightarrow}}{x}, \alpha, \overset{\overset{w[1:i]}{\Longrightarrow}}{x\alpha})$ is an edge of $STree'(w[1:i])$ where $\overset{\overset{w[1:i]}{\Longrightarrow}}{x\alpha}$ 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,

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

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

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

Let $(\overset{\overset{w[1:i]}{\Longrightarrow}}{x}, \alpha)$ be the canonical reference pair for the active point, namely, $\overset{\overset{w[1:i]}{\Longrightarrow}}{x} \cdot \alpha = w[\ell : i]$. Focus on the edge $(\overset{\overset{w[1:i]}{\Longrightarrow}}{x}, \alpha\beta, \overset{\overset{w[1:i]}{\Longrightarrow}}{x\alpha\beta})$ where $\beta \neq \varepsilon$. The edge is replaced by the edges $(\overset{\overset{w[1:i+1]}{\Longrightarrow}}{x}, \alpha, \overset{\overset{w[1:i+1]}{\Longrightarrow}}{x\alpha})$ and $(\overset{\overset{w[1:i+1]}{\Longrightarrow}}{x\alpha}, \beta, \overset{\overset{w[1:i+1]}{\Longrightarrow}}{x\alpha\beta})$ where $\overset{\overset{w[1:i+1]}{\Longrightarrow}}{x\alpha}$ is a new explicit node. Then a new edge $(\overset{\overset{w[1:i+1]}{\Longrightarrow}}{x\alpha}, \gamma, \overset{\overset{w[1:i+1]}{\Longrightarrow}}{x\alpha\gamma})$ is created, where $\gamma = w[i+1:i+1]$. Note $\overset{\overset{w[1:i+1]}{\Longrightarrow}}{x\alpha\gamma} = w[\ell : i+1]$. 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 $\overset{w[1:i+1]}{\overset{\Longrightarrow}{x\alpha}}$. Thus we once move to its parent node $\overset{w[1:i+1]}{\overset{\Longrightarrow}{x}}$ for which $\mathit{suf}(\overset{w[1:i+1]}{\overset{\Longrightarrow}{x}})$ must have already been computed. Let $\mathit{suf}(\overset{w[1:i+1]}{\overset{\Longrightarrow}{x}}) = \overset{w[1:i+1]}{\overset{\Longrightarrow}{y}}$, where there exists some character a such that $\overset{w[1:i+1]}{\overset{\Longrightarrow}{x}} = a \cdot \overset{w[i+1]}{\overset{\Longrightarrow}{y}}$. Note that $\overset{w[1:i+1]}{\overset{\Longrightarrow}{y}} \cdot \alpha = w[\ell - 1 : i]$. We go down from the node $\overset{w[1:i+1]}{\overset{\Longrightarrow}{y}}$ 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 $\mathit{suf}(\overset{w[1:i+1]}{\overset{\Longrightarrow}{x\alpha}})$ is then set to $w[\ell - 1 : i]$. This way the algorithm 'simulates' the suffix-link-traversal of suffix tries.

Figure 4.2 shows the on-line construction of *STree*$'$(cocoa).

A pseudo-code for Ukkonen's algorithm is shown in Fig 4.3. 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 6 (Ukkonen [73]) *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.*

4.3 On-Line Construction of CDAWGs

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 4.4 that shows the on-line construction of *CDAWG*$'$(cocoa), in comparison with Figure 4.2 displaying the on-line construction of *STree*$'$(cocoa). Compare *CDAWG*$'$(co) and *STree*$'$(co). While strings co and o are separately represented in *STree*$'$(co), they are in the same node in *CDAWG*$'$(co). The 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*$'$(coc) and *CDAWG*$'$(coco)).

Focus on the first step of the update of *CDAWG*$'$(coco) to *CDAWG*$'$(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*$'$(cocoa), edge $(\varepsilon, \texttt{ocoa}, \texttt{ocoa})$

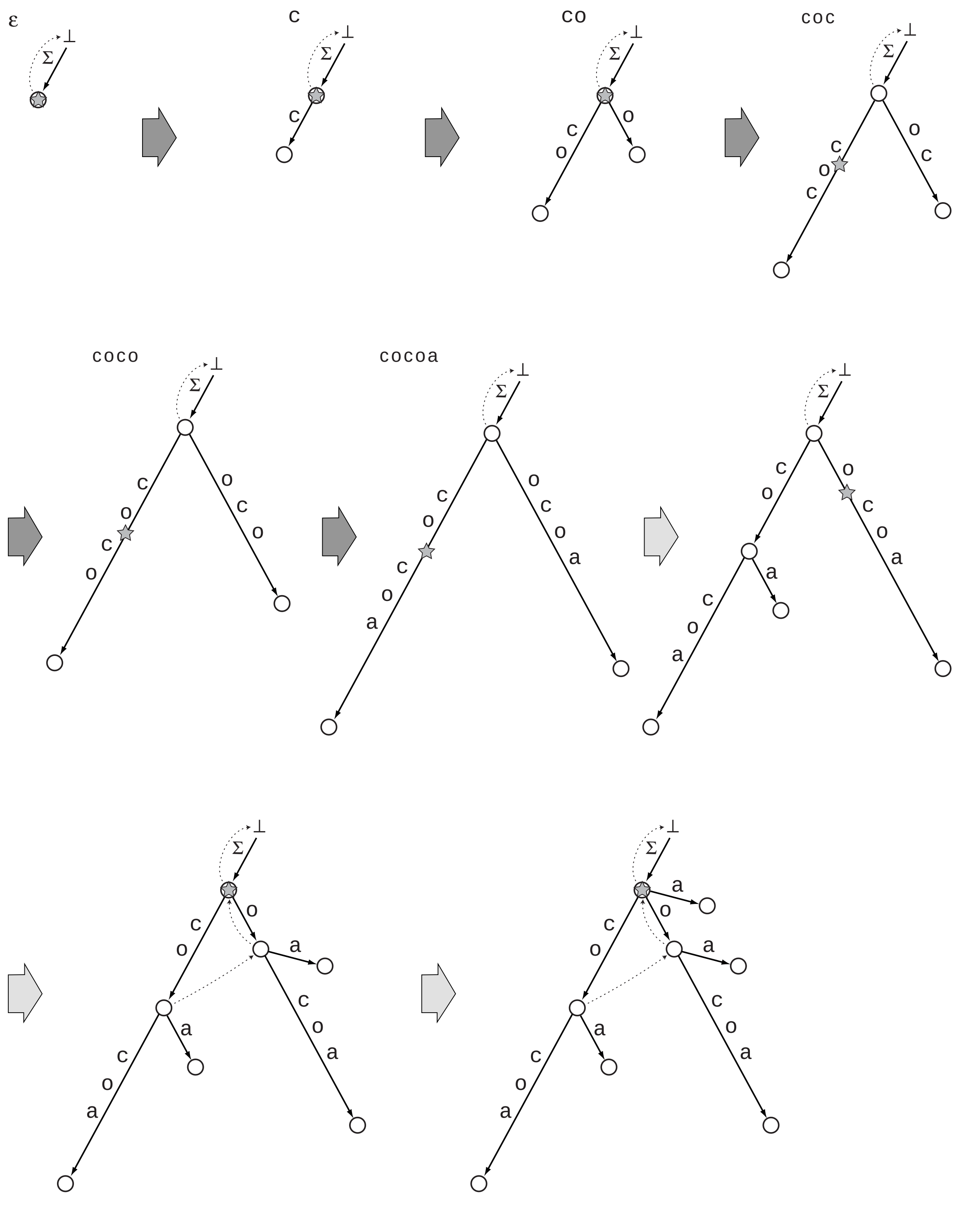


Figure 4.2: On-line construction of $STree'(w)$ with $w = \texttt{cocoa}$. The star represents the active point for each step.

```
Algorithm for on-line construction of $STree'(w\$)$
in alphabet $\Sigma = \{w[-1], w[-2], \ldots w[-m]\}$.
/* $ 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 $k \leq p-1$ 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 $k \leq p$ 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 $p' - k' \leq p - k$ do
 4     $k := k + p' - k' + 1$;    $s := s'$;
 5     if $k \leq p$ 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 4.3: Ukkonen's on-line algorithm for constructing suffix trees.

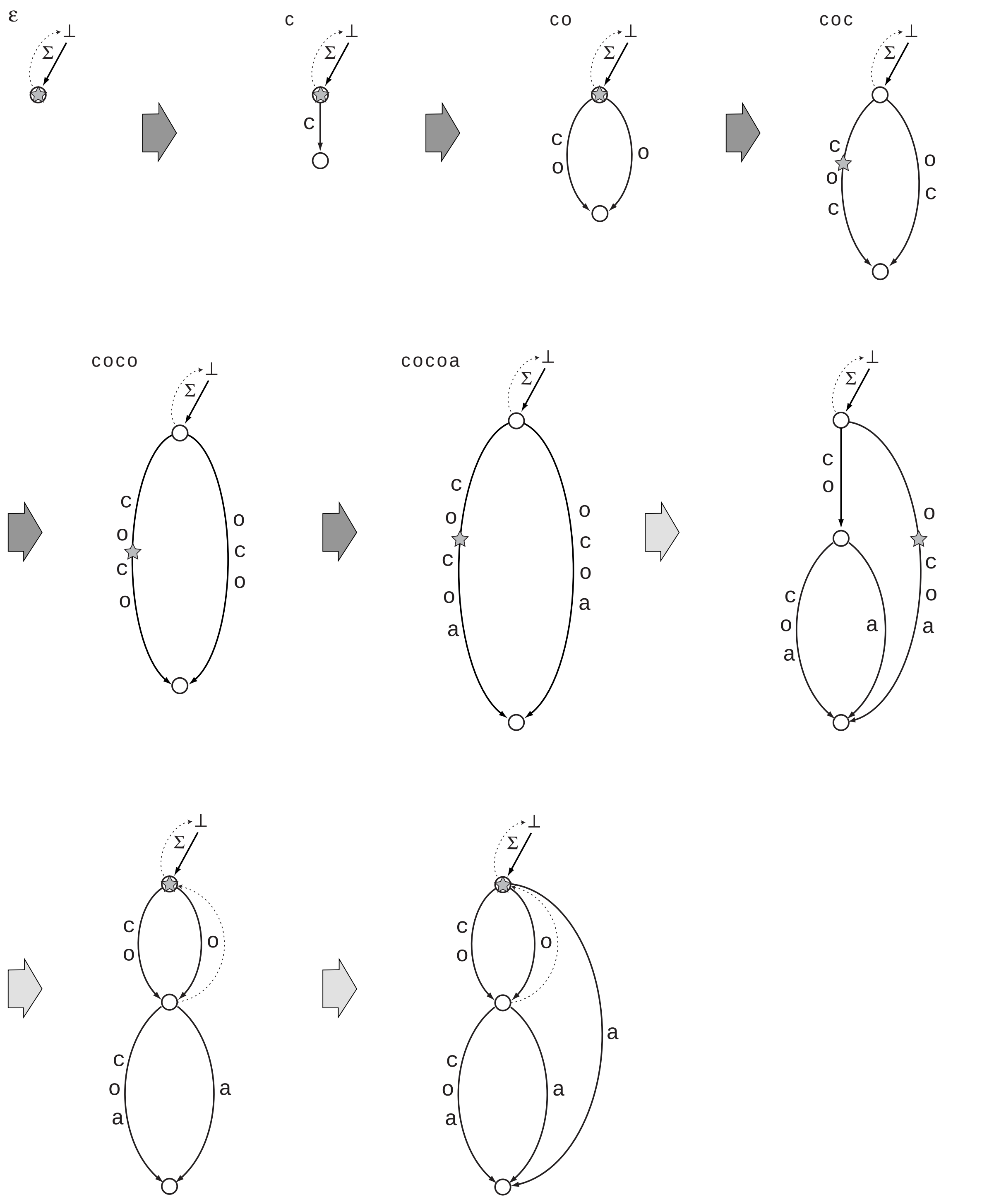


Figure 4.4: On-line construction of $CDAWG'(w)$ with w = cocoa. The star mark represents the active point for each step.

is split into two edges $(\varepsilon, \mathtt{o}, \mathtt{o})$ and $(\mathtt{o}, \mathtt{coa}, \mathtt{ocoa})$, and then an open edge $(\mathtt{o}, \mathtt{a}, \mathtt{oa})$ is newly created. However, in case of the CDAWG, edge $(\varepsilon, \mathtt{ocoa}, \mathtt{ocoa})$ is *redirected* to node $\mathtt{co}$, and the label is simultaneously modified accordingly. Since strings $\mathtt{co}$ and $\mathtt{o}$ are equivalent under the equivalent relation $\equiv^R_{\mathtt{cocoa}}$, they are *merged* into a single node in $\mathit{CDAWG}'(\mathtt{cocoa})$.

4.3.1 The Algorithm

The algorithm presented in this section for 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 $\mathit{CDAWG}'(w)$ 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 $\mathit{CDAWG}'(u)$ to $\mathit{CDAWG}'(ua)$. The equivalence test is performed on the basis of Lemma 5 to be given in the sequel.

- Consider strings $x, y \in \mathit{Substr}(u)$ 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 $\mathit{CDAWG}'(u)$. Note that, however, x, y might not be equivalent under $\equiv^R_{ua}$. When $\mathit{CDAWG}'(u)$ is updated to $\mathit{CDAWG}'(ua)$, then the node has to be *separated* into two nodes $[x]^R_{ua}$ and $[y]^R_{ua}$. Since this node separation happens only when $x \notin \mathit{Suffix}(ua)$ but $y \in \mathit{Suffix}(ua)$, we can do this procedure after we find the end point. The condition of the node separation will be given later on, in Lemma 6.

Merging Implicit Nodes.

As mentioned above, it can happen that two or more nodes implicit in $\mathit{CDAWG}'(u)$ are merged into one explicit node in $\mathit{CDAWG}'(ua)$. As a concrete example, we show in Figure 4.6 the snapshot of the conversion of $\mathit{CDAWG}'(u)$ into $\mathit{CDAWG}'(ua)$ with $u = \texttt{abcabcab}$ and $a = \texttt{a}$.

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

Lemma 5 *Let $w \in \Sigma^*$. For any strings $x, y \in \mathit{Substr}(w)$ with $y \in \mathit{Suffix}(x)$,*

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

Proof. If $x \equiv^R_w y$, we have $\overset{w}{\overleftarrow{x}} = \overset{w}{\overleftarrow{y}}$ by Definition 3. By Corollary 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 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 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 7. 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 $x \equiv^R_w y$. □

For any string $x \in \mathit{Substr}(w)$, 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 5.

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 $[\overset{w}{\overset{\Longrightarrow}{x}}]^R_w$ instead

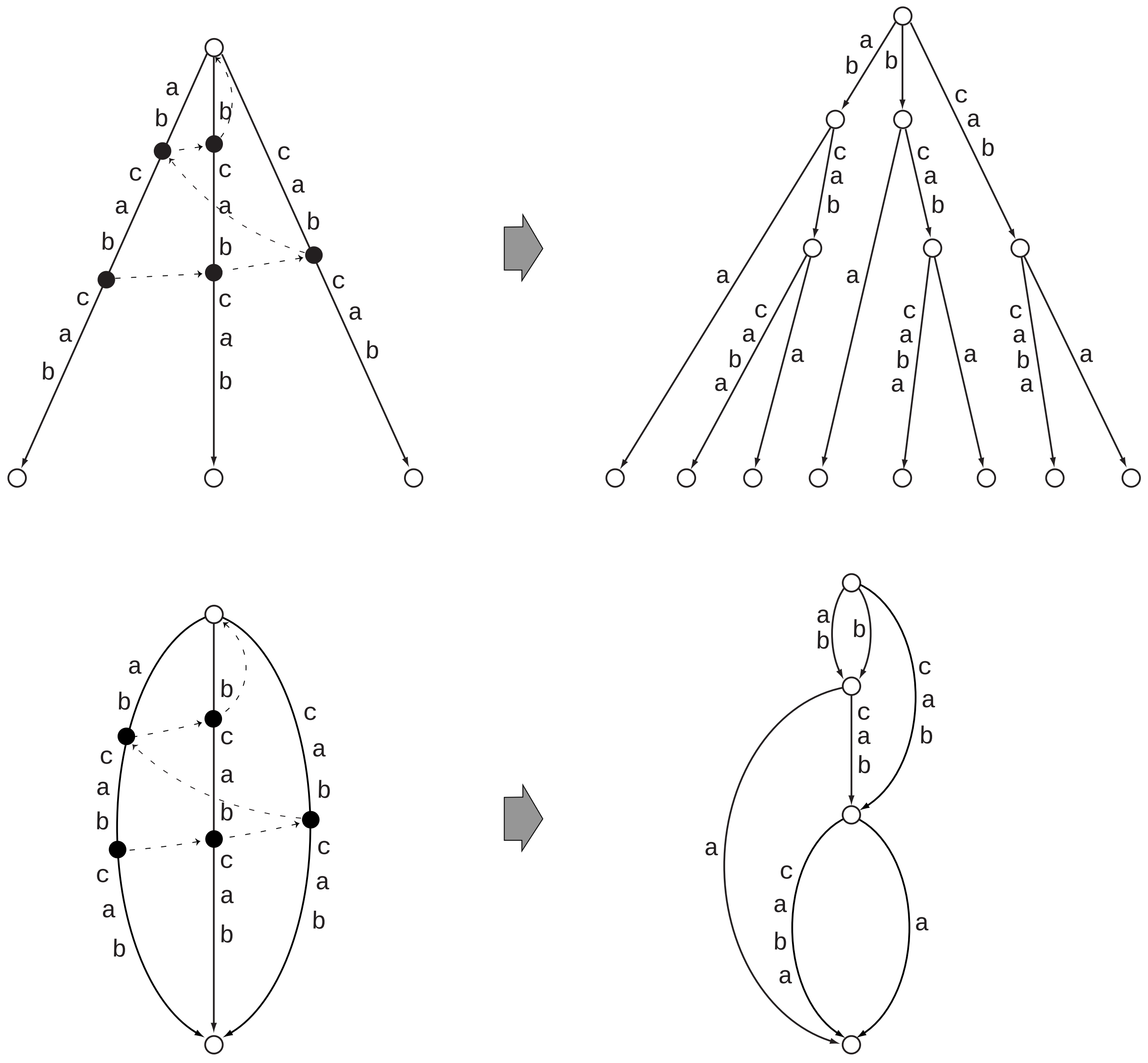


Figure 4.5: Comparison of conversions. One is from $STree'(u)$ to $STree'(ua)$, while the other is from $CDAWG'(u)$ to $CDAWG'(ua)$ for u = `abcabcab` and a = `a`. The black circles represent implicit nodes to be merged in the next step, connected by implicit suffix links corresponding to the traversal by the active point.

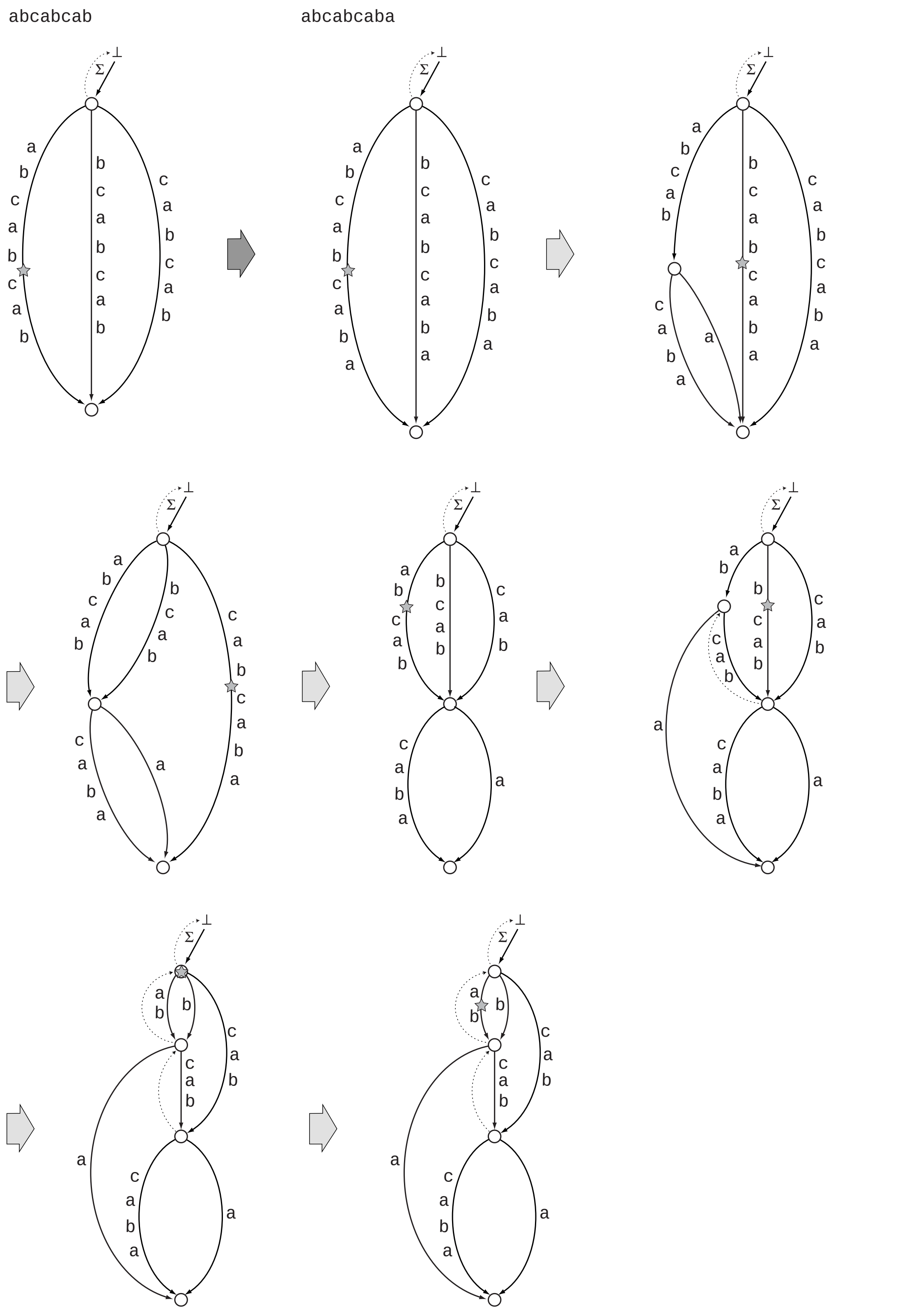


Figure 4.6: Detailed conversion from $CDAWG'(u)$ to $CDAWG'(ua)$ for $u =$ abcabcab and $a =$ a.

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 4.5, where $u = \texttt{abcabcab}$. The black points are the implicit nodes the active point traverses in the next step via 'implicit' suffix links. In $\mathit{CDAWG}'(u)$, $[\overset{u}{\overrightarrow{\texttt{cab}}}]^R_u = [\overset{u}{\overrightarrow{\texttt{ab}}}]^R_u = [\overset{u}{\overrightarrow{u}}]^R_u$. However, in $\mathit{CDAWG}'(ua)$, $\texttt{cab} \not\equiv^R_{ua} \texttt{ab}$ where $a = \texttt{a}$. See Figure 4.6 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 $[\overset{u}{\overrightarrow{x}}]^R_u$ with $x \in \mathit{Substr}(u)$ might be separated into two explicit nodes $[\overset{ua}{\overrightarrow{x}}]^R_{ua}$ and $[\overset{ua}{\overrightarrow{y}}]^R_{ua}$ 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 [9]. 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 6 (Blumer et al. [9]) *Let $u \in \Sigma^*$ and $a \in \Sigma$. Let z be the LRS of ua. For a string $x \in \mathit{Substr}(u)$, assume $x = \overset{u}{\overleftarrow{x}}$. Then,*

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

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 $y \in \mathit{Substr}(u)$ 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,

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

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

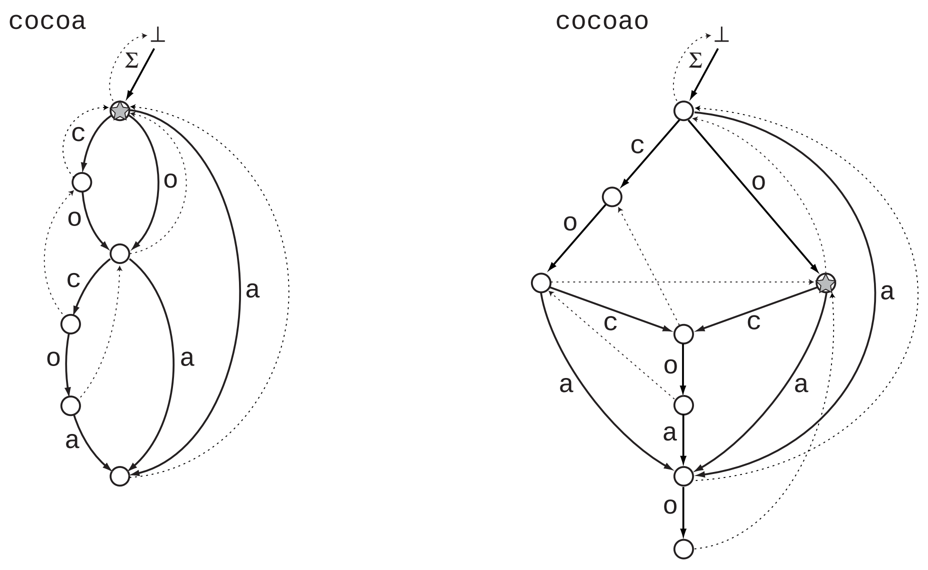


Figure 4.7: The update of *DAWG*(u) to *DAWG*(ua), where u = cocoa and a = o.

Figure 4.7 shows the conversion from *DAWG*(u) to *DAWG*(ua) with u = cocoa and a = a. The LRS of the string cocoao is o, therefore we focus on edge $([\varepsilon]^R_u, \mathtt{o}, [\mathtt{o}]^R_u)$. Since $\mathit{length}([\varepsilon]^R_u) + |\mathtt{o}| = 1 < \mathit{length}([\mathtt{o}]^R_u) = 2$, node $[\mathtt{o}]^R_u$ is separated into two nodes $[\mathtt{co}]^R_{ua}$ and $[\mathtt{o}]^R_{ua}$, as shown in Figure 4.7.

Now we go back to the update of *CDAWG*′(u) to *CDAWG*′(ua). The test of whether to separate a node when a CDAWG is updated can also be done on the basis of Lemma 6 in the very similar way. Since only explicit nodes can be separated, we merely 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 7 *Let $w \in \Sigma^*$. Assume the LRS of w is z. Then, if $z = \overset{w}{\overrightarrow{z}}$, $\overset{w}{\overrightarrow{x}} = \overset{w}{\overleftrightarrow{x}}$ for any string $x \in Substr(w)$.*

This lemma guarantees that the representative of $[\overset{u}{\overrightarrow{x}}]^R_u$ is equal to $\overset{u}{\overleftrightarrow{x}}$ if the preconditions 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 *CDAWG*′(u) is updated to *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 $y \in Substr(u)$ be the string such that $\overset{u}{\overleftrightarrow{y}} \cdot \alpha = z$ for some string $\alpha \in Substr(u)$. Note

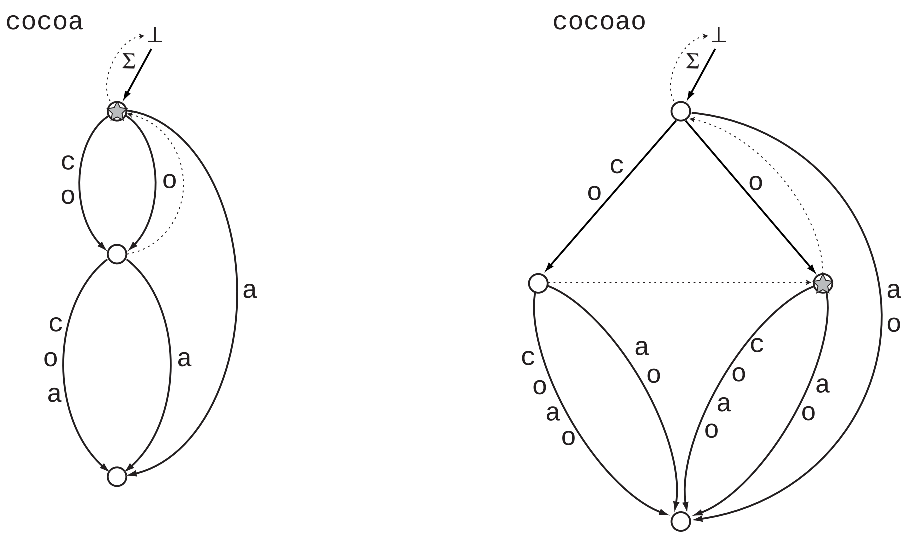


Figure 4.8: The update of $CDAWG'(u)$ to $CDAWG'(ua)$, where $u = \texttt{cocoa}$ and $a = \texttt{o}$.

that there then exists an edge $([y]^R_u, \alpha, [x]^R_u)$. Then,

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

Figure 4.8 shows the update of $CDAWG'(u)$ to $CDAWG'(ua)$, where $u = \texttt{cocoa}$ and $a = \texttt{o}$. The LRS of the string cocoao is o, therefore we focus on edge $([\overset{u}{\overset{\Longrightarrow}{\varepsilon}}]^R_u, \texttt{o}, [\overset{u}{\overset{\Longrightarrow}{\texttt{o}}}]^R_u)$. Since $\mathit{length}([\overset{u}{\overset{\Longrightarrow}{\varepsilon}}]^R_u) + |\texttt{o}| = 1 < \mathit{length}([\overset{u}{\overset{\Longrightarrow}{\texttt{o}}}]^R_u) = 2$, node $[\overset{u}{\overset{\Longrightarrow}{\texttt{o}}}]^R_u$ is separated into two nodes $[\overset{ua}{\overset{\Longrightarrow}{\texttt{co}}}]^R_{ua}$ and $[\overset{ua}{\overset{\Longrightarrow}{\texttt{o}}}]^R_{ua}$, as shown in Figure 4.8.

Pseudo-Code.

The algorithm is described in Figure 4.9 and Figure 4.10. 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 $DAWG(w)$ given by Blumer et al. [9], 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 $k \le p - 1$ 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 4.9: 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 $k' \le p$ 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 $s'$;    /* 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 $(s', k') \ne canonize(s, (k, p))$;
13 return $(r', p + 1)$;
```

Figure 4.10: 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. 4.3, they are omitted.

Complexity of the Algorithm.

Theorem 7 *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 [9] and the one of the on-line algorithm for suffix trees [73]. 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 $x \in Substr(w)$. 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 [9]. 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 [9], 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 [73]). 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. □

Chapter 5

CDAWGs for Sets of Strings

5.1 Construction of CDAWGs for Sets of Strings

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

Blumer et al. [10] introduced DAWGs for sets of strings, and presented an algorithm that builds $\mathit{DAWG}(S)$ in $O(\|S\|)$ time. They also introduced CDAWGs for sets of strings. Their algorithm for construction of $\mathit{CDAWG}(S)$ runs in linear time in the input size (namely, in $O(\|S\|)$ time), but not in time linear in the output size because it first builds $\mathit{DAWG}(S)$ and then converts it to $\mathit{CDAWG}(S)$ by deleting internal nodes of out-degree one and concatenating their edges accordingly. Kosaraju [46] introduced the suffix tree for set S of strings, often referred to the *generalized suffix tree* for S. A slight modification of Ukkonen's algorithm recalled in Section 4.2 is capable of constructing $\mathit{STree}'(S)$ in $O(\|S\|)$ time.

In this section, we give the first algorithm which builds $\mathit{CDAWG}'(S)$ in $O(\|S\|)$ time, and directly. Let $S = \{w_1, w_2, \dots, w_k\}$, where $k = |S|$. Then we consider set $S' = \{w_i\$_i \mid w_i \in S \text{ and } \$_i \notin \mathit{Substr}(S) \text{ for } 1 \leq i \leq |S|\}$. Notice that S' has the prefix property, and thus, $\mathit{CDAWG}'(S') = \mathit{CDAWG}(S')$ for any S. $\mathit{CDAWG}'(S')$ can be constructed by a slight modification of the algorithm proposed in the previous section. We use a global variable e_i for each string in S', where $1 \leq i \leq |S|$, which indicates the ending position of open edges for each string. We treat 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

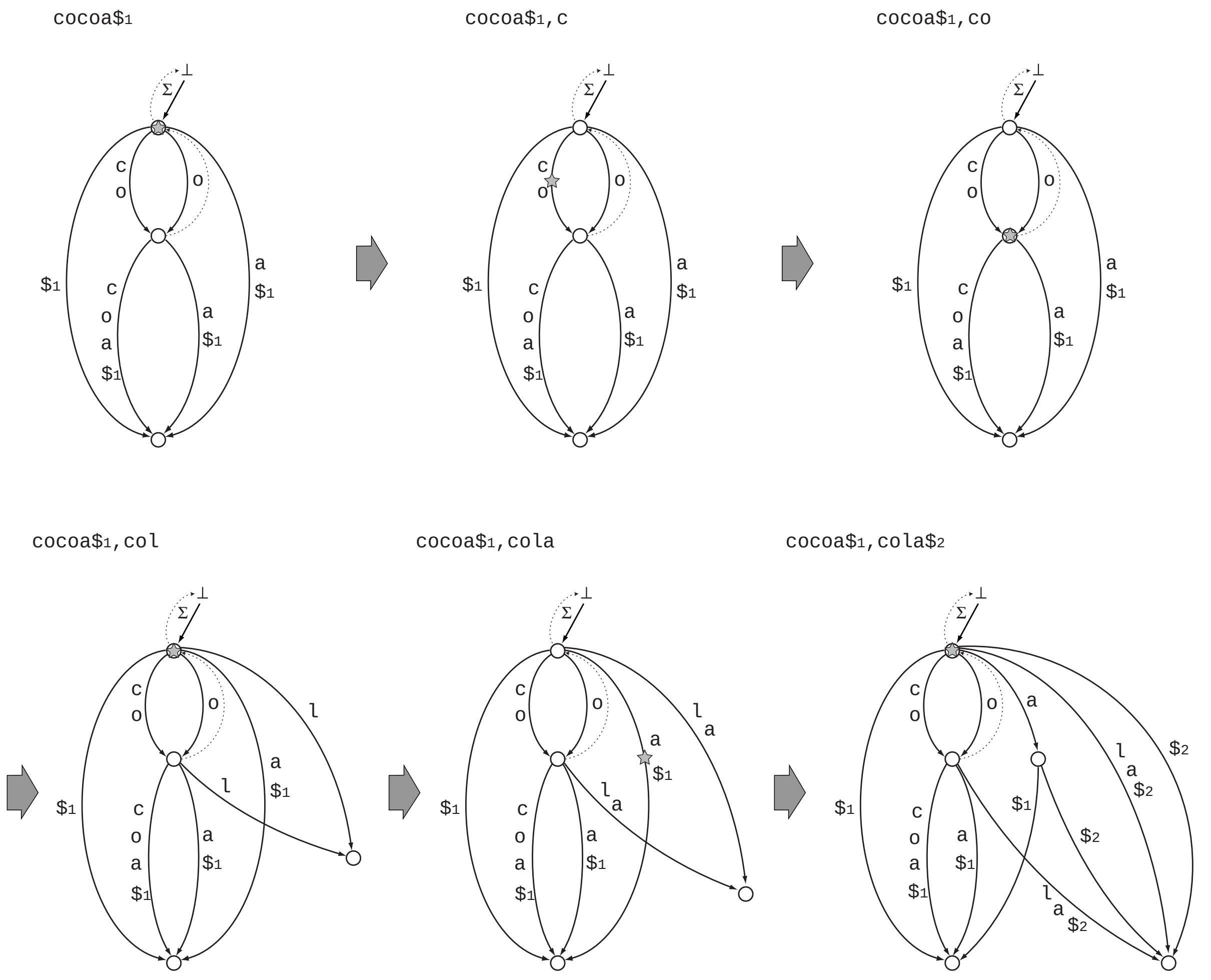


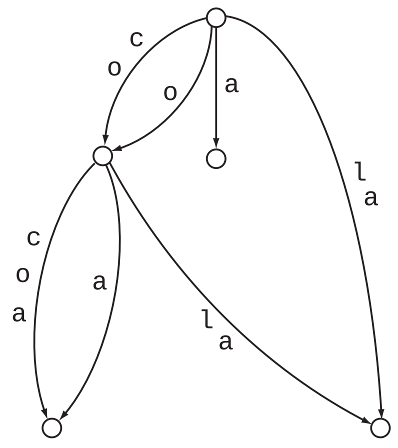
Figure 5.1: Construction of *CDAWG*$'(S')$ for $S' = \{\texttt{cocoa}\$_1, \texttt{cola}\$_2\}$.

character is scanned. Thereby we have the following.

Theorem 8 *Assume* Σ *is a fixed alphabet. For any set* S *of strings, the proposed algorithm constructs CDAWG*$'(S')$ *on-line and in* $O(\|S'\|)$ *time, using* $O(\|S'\|)$ *space.*

Figure 5.1 shows construction of *CDAWG*$'(S')$, where $S' = \{\texttt{cocoa}\$_1, \texttt{cola}\$_2\}$.

Remark. As a secondary effect of the end-markers $\$_i$, we obtain a good feature on *CDAWG*$'(S')$. For $S = \{\texttt{cocoa}, \texttt{cola}\}$, *CDAWG*$(S)$ is shown in Figure 5.2. One can see there are *three* sink nodes though S contains only *two* strings in it. Because `a` is a substring of both strings in S, it cannot belong to only one of another two sink nodes which individually correspond to `cocoa` and `cola`. This case, we cannot readily specify what string(s) in S the string `a` is a substring of. However, there is no difficulty to specify



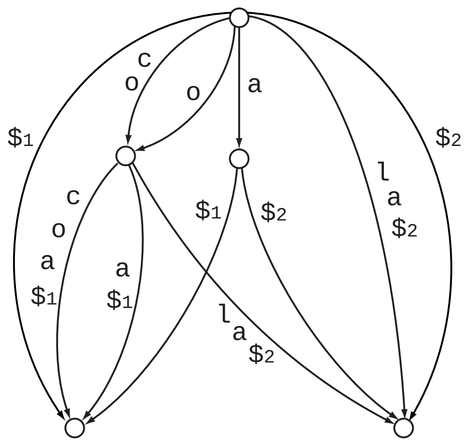


Figure 5.2: For set $S = \{\texttt{cocoa}, \texttt{cola}\}$, *CDAWG*$(S)$ is shown on the left. For set $S' = \{\texttt{cocoa}\$_1, \texttt{cola}\$_2\}$, *CDAWG*$'(S')$ is displayed on the right.

it in *CDAWG*$'(S')$, since there exactly exist $|S'| = |S|$ sink nodes in it (see the right of Figure 5.2).

5.2 Constructing the CDAWG for a Trie

In this chapter, we consider the case that the set S is given in the form of a trie (see Section 2.3.1). Namely, our input is *Trie*(S) and output is *CDAWG*$'(S)$. Let $\|S\| = \ell$. Since the trie shares common prefixes of strings in *Substr*(S), in general the number N of nodes of the trie is less than ℓ. We show a non-trivial extension of the algorithm of the previous section, which constructs the CDAWG for a trie in $O(N)$ time and space.

Some related work can be seen in literature: Kosaraju [46] introduced the suffix tree for a *reversed* trie, where common suffixes of strings in *Substr*(S) are merged together. We denote it by *Trie*$^{\text{rev}}(S)$. Let M be the number of nodes in *Trie*$^{\text{rev}}(S)$. Kosaraju showed an algorithm to construct *STree*(S) in $O(M \log M)$ time. Later on, Breslauer [11] improved the time complexity to $O(M)$.

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 *Trie*(S) is more natural than given as *Trie*$^{\text{rev}}(S)$.

This result was originally published in [36].

5.2.1 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 20 $\mathit{Trie}^{\mathrm{rev}}(S)$ 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 21 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 8 *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 [11] traverses a given reversed trie from a leaf node, it is not eligible for constructing the suffix tree for a set of strings that does not have the suffix property.

Proposition 9 *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 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. 5.3.

Theorem 9 (Breslauer [11]) *If S'' has the suffix property, $\mathit{STree}(S'')$ can be constructed in $O(N)$ time, where N is the number of nodes in $\mathit{Trie}^{\mathrm{rev}}(S'')$.*

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

Proposition 10 *Given a set $S = \{w_1, \dots, w_k\}$ with $k \ge 1$, let $S' = \{w_1\$_1, \dots, w_k\$_k\}$. Then, $\mathit{Trie}(S')$ has at most $\|S'\| + 1 = \|S\| + |S| + 1$ nodes.*

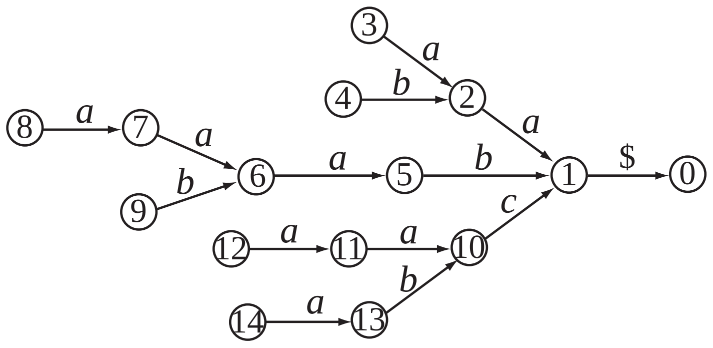


Figure 5.3: $\mathit{Trie}^{\mathrm{rev}}(S'')$ for $S'' = \{aaab\$, aac\$, aa\$, abc\$, bab\$, ba\$\}$.

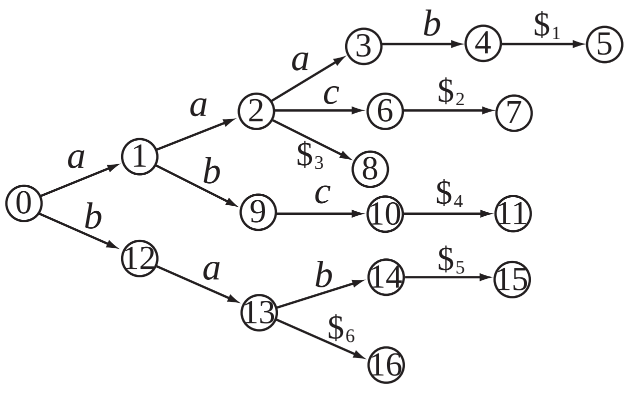


Figure 5.4: $\mathit{Trie}(S')$ for $S' = \{aaab\$_1, aac\$_2, aa\$_3, abc\$_4, bab\$_5, ba\$_6\}$.

Figure 5.4 shows $\mathit{Trie}(S')$ with $S' = \{aaab\$_1, aac\$_2, aa\$_3, abc\$_4, bab\$_5, ba\$_6\}$. 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. 5.4.

$\mathit{Trie}(S')$ is the input of our algorithm to be introduced in the next section.

5.2.2 The Algorithm

We firstly note that the CDAWG for $\mathit{Trie}(S')$ is the same as the CDAWG of S', except the following point. The label of an edge in $\mathit{CDAWG}(S')$ 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, the corresponding edge of the CDAWG for $\mathit{Trie}(S')$ refers to a pair of nodes in $\mathit{Trie}(S')$, 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 over *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 Figure 5.2.2.

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

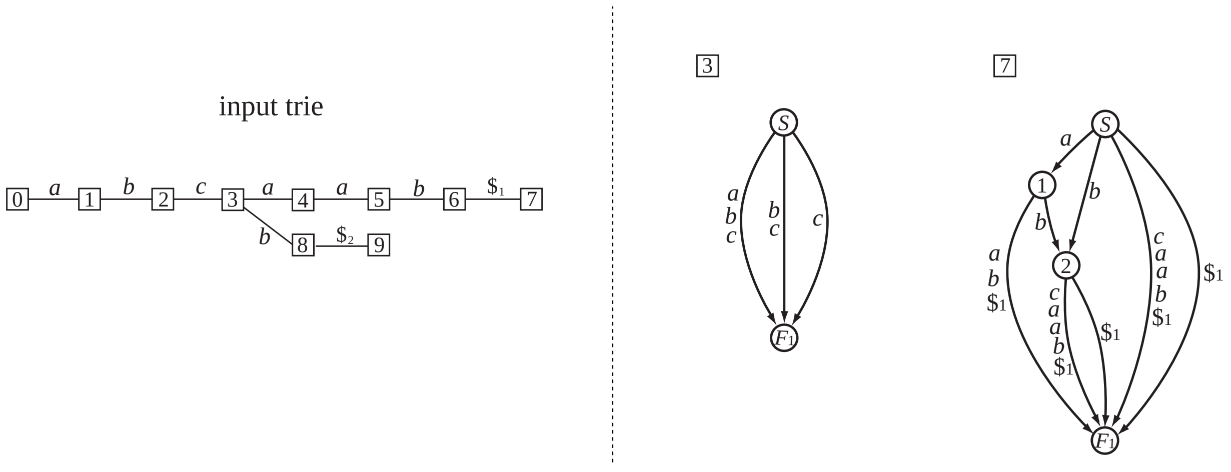


Figure 5.5: *Trie*(S') for $S = \{abcaab\$_1, abcb\$_2\}$ is shown left. When the algorithm focuses on node $\boxed{3}$ in *Trie*(S'), it needs to memorize the location in the CDAWG corresponding to *abc*. Since there is no node but F_1 at the location, it is memorized by a reference pair $(0, abc)$. 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 $\boxed{3}$.

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 shall 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 Figure 5.6. The variable *current_node* indicates the node on which the algorithm currently focuses in *Trie*(S'). The variable *advanced_point* is of the 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*, **function** *update_CDAWG* updates the CDAWG with a letter c. **function** *update_CDAWG* is essencially the same as the one for construction of the CDAWG for a set of strings directly, previously introduced in Section 5.1, excepting that *update_CDAWG* creates a new edge stemming from the node latest created

```
Algorithm to construct the CDAWG for $Trie(S')$    /* The input is $Trie(S')$ */
 1 $current\_node := root$;    /* the root node of $Trie(S')$ */
 2 $active\_point := (source, \varepsilon)$;
 3 $advanced\_point := (source, \varepsilon)$;
 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 $|label\_set| \geq 2$ then $create\_node(advanced\_point)$;
 4 for each $c \in label\_set$ 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 5.6: Algorithm to construct the CDAWG for a trie.

by **function** *create_node*.

An example of the construction of the CDAWG for a trie is shown in Figure 5.2.2.

Finally, we have the following theorem.

Theorem 10 *$CDAWG'(S')$ can be constructed in $O(N)$ time, where N is the number of nodes in $Trie(S')$.*

Proof. We first explain that the modification of **function** *update_CDAWG* and **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. Therefore, **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 **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, **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. □

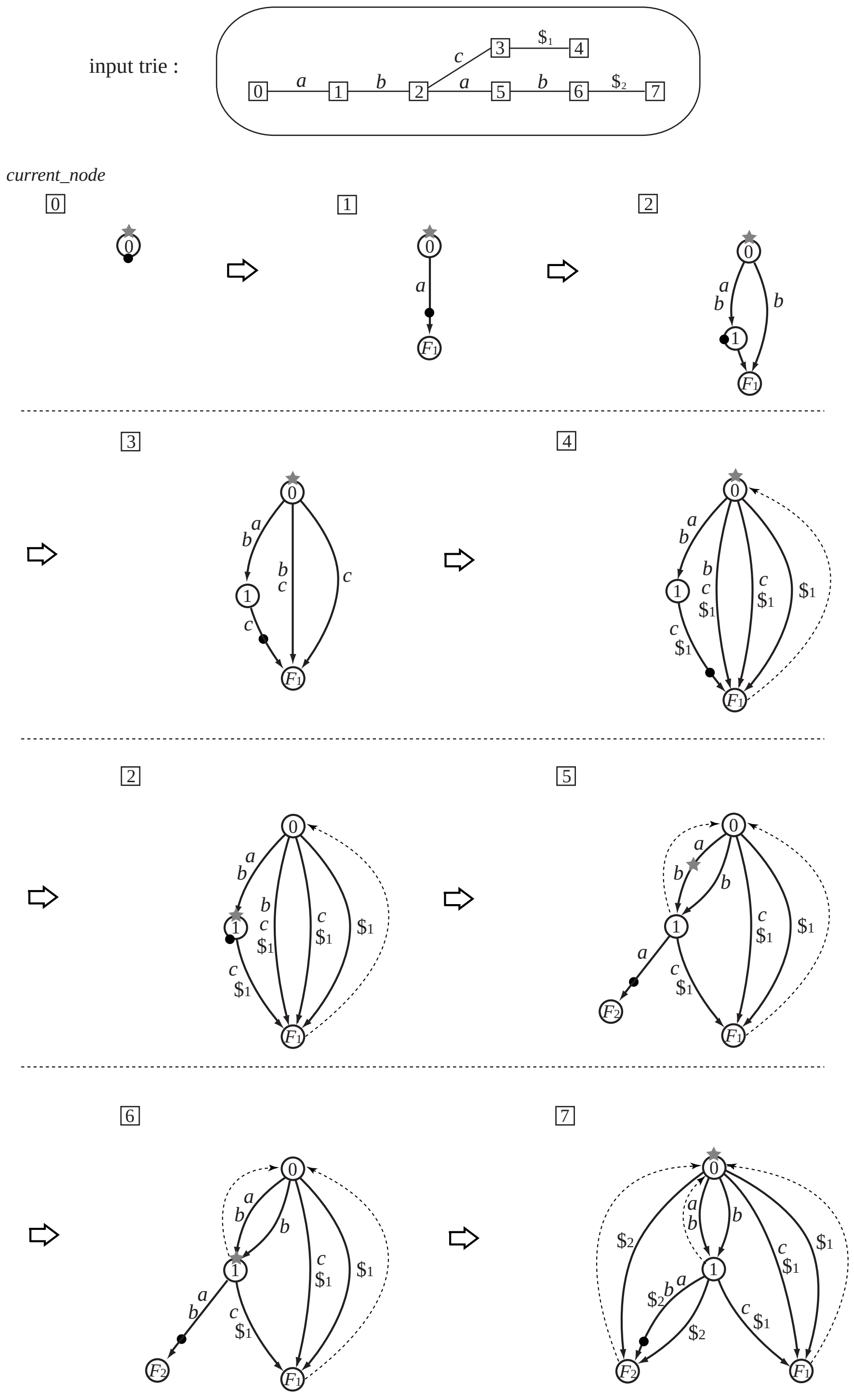


Figure 5.7: Construction of the CDAWG for $Trie(S')$, where $S' = \{abc\$_1, abab\$_2\}$. The gray starred point represents *active_point*, and the black dotted point represents *advanced_point*. For simplicity, the bottom node is omitted. As node $\boxed{2}$ in the trie is branching, a new node $\boxed{1}$ 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* $= 1$.

Chapter 6

CDAWGs for a Sliding Window

In such a situation that only a limited amount of memory is available to us, such as when processing an on-line data stream, the *sliding window mechanism* is very useful and often applied actually. There we construct and maintain an index structure for a window of width M that slides over w. This case, therefore, we have to update the index structure dynamically.

Another application of the sliding window mechanism is the *prediction by partial matching* (*PPM*) style statistical data compression model [14, 57]. PPM* [13] is an improvement that allows unbounded context length. PPM* employs a tree structure called the context trie, which supports indices of the *whole* input string. The drawback of PPM* is, however, its too much computational resources in both time and space, which weakens its practical usefulness. In particular, the context trie occupies major part of the space requirement, since it inherently has the same structure as the suffix trie requiring quadratic space. Since PPM is known to be the best text compression scheme from the viewpoint of compression ratio, it is quite important and meaningful to reduce space requirement of PPM.

Larsson's suffix tree for a sliding window offered a variant of PPM*, feasible in practice since its space requirement is linear in the window size M and the running time is linear in the length of the input string w [49]. Assume that the window is now covers a substring u of w such that $|u| = M \geq 1$. That is, we now have $STree'(u)$. We then slide the window from left to right. To do so, we first need to add $a \in \Sigma$ to the right of u, where a is the character immediately following u in the current position of the window over w. It implies we have to update $STree'(u)$ to $STree'(ua)$, and it can be done by Ukkonen's on-line algorithm [73]. Thus, the key is how to convert $\mathit{STree}'(ua)$ into $\mathit{STree}'(va)$, where $v \in \Sigma^*$ and $bv = u$ with some $b \in \Sigma$. Larsson showed how to manage the above operation in amortized constant time.

In this chapter, we take another approach to reduce the space requirement in PPM*-style statistical compression. We propose an algorithm to maintain *CDAWGs* for a sliding window, which performs in $O(|w|)$ time using $O(M)$ space. Recall CDAWGs require less space than suffix trees in both theory and practice [10, 18]. In Chapter 4, we presented an on-line algorithm that constructs CDAWGs in linear time and space. Moving the rightmost position of a sliding window can be accomplished by this algorithm adding new forthcoming characters to the right of the current string as well as the case of suffix trees. Hence, again the key point of the algorithm is how to move the left most position of the sliding window.

In case of a suffix tree, it is also rather straightforward to advance the leftmost position of a sliding window: basically we have only to remove the leaf node and its in-coming edge corresponding to the longest suffix. However, since a CDAWG is a graph, the matter is much more complex and technically difficult. Thus more detailed and precise discussions are necessary. In addition, we have to ensure that no edge labels refer to positions outside a sliding window. To guarantee it, Larsson utilized the technique of *credit issuing* first introduced in [22], which takes amortized constant time. We introduce an extended version of credit issuing that is modified to be suitable for treating CDAWGs for a sliding window.

The result was presented originally in [41].

6.1 Suffix Trees for a Sliding Window

Larsson [49] introduced an algorithm for maintaining suffix trees for a *sliding window*, whose width is $M \geq 1$. Let i (resp. j) be the leftmost (resp. rightmost) position of the window sliding over w, that is, $j - i + 1 = M$. To move the sliding window ahead, we need to increment i and j. Incrementing j can be accomplished by Ukkonen's on-line algorithm. On the other hand, incrementing i means to delete the leftmost character of the currently scanned string, that is, to convert $\mathit{STree}'(bu)$ into $\mathit{STree}'(u)$ with some $b \in \Sigma$ and $u, bu \in \mathit{Substr}(w)$. We focus on the path of $\mathit{STree}'(bu)$ which spells out bu from the root node. This path is called the *backbone* of $\mathit{STree}'(bu)$. Let x be the longest string in

Prefix$(bu) - \{bu\}$ such that $\overset{bu}{\overrightarrow{x}} = x$. The locus of x in *STree*$'(bu)$ is called the *deletion point* and denoted by *DelPoint*(bu). On the other hand, let z be the longest string in *Prefix*$(bu) - \{bu\}$ such that $\overset{bu}{\Rightarrow}\!\!\!\!\!z = z$. The string z is called the *last node* in the backbone and denoted by *LastNode*(bu).

When *DelPoint*$(bu) =$ *LastNode*(bu), there is an explicit node representing the string x in *STree*$'(bu)$. Then there exists an edge $(\overset{bu}{\overrightarrow{\overrightarrow{x}}}, y, \overset{bu}{\overrightarrow{\overrightarrow{bu}}})$ in *STree*$'(bu)$ where $xy = bu$. Only by removing this edge, we can obtain *STree*$'(u)$.

When *DelPoint*$(bu) \neq$ *LastNode*(bu), it follows from Proposition 3 that $x \in$ *Suffix*(bu). Moreover, $x =$ *LRS*(bu), as to be proven by Lemma 10 in Section 6.2.1. Namely, the active point is on the locus for x in *STree*$'(bu)$. Let $(\overset{bu}{\overrightarrow{\overrightarrow{s}}}, y, \overset{bu}{\overrightarrow{\overrightarrow{bu}}})$ be the edge on which x is represented. Let $\overset{bu}{\overrightarrow{\overrightarrow{s}}} \cdot t = x$, where $t \in$ *Prefix*(y). We *shorten* the edge to $(\overset{u}{\overrightarrow{\overrightarrow{s}}}, t, \overset{u}{\overrightarrow{\overrightarrow{x}}})$, and move the active point to the locus for the one-character shorter suffix of x.

Theorem 11 ([49]) *Let $w \in \Sigma^*$ and M be the window size. Larsson's algorithm runs in $O(|w|)$ time using $O(M)$ space.*

6.2 CDAWGs for a Sliding Window

In this section, we consider maintaining a CDAWG for a sliding window. Advancing the rightmost position of the window can be done by the on-line algorithm proposed in Chapter 4. Thus the matter is to move ahead the leftmost position of the window.

6.2.1 Edge Deletion

Given *CDAWG*$'(w)$, we also focus on its backbone, the path spelling out w from the source node. Let $x =$ *DelPoint*(w). If *DelPoint*$(w) =$ *LastNode*(w), we remove the edge $([\overset{w}{\overrightarrow{\overrightarrow{x}}}]^R_w, y, [\overset{w}{\overrightarrow{\overrightarrow{w}}}]^R_w)$ such that $xy = w$. However, notice that this method might remove other suffixes of w from the CDAWG. More precise arguments follow.

Lemma 8 *Let $w \in \Sigma^+$, $x =$ DelPoint(w), and $z =$ LastNode(w). Assume $x = z$. Let s be any string in $[\overset{w}{\overrightarrow{\overrightarrow{x}}}]^R_w = [\overset{w}{\overrightarrow{\overrightarrow{z}}}]^R_w$. Then there uniquely exists a string $y \in \Sigma^+$ such that $sy \in$ Suffix(w).*

Proof. Since $x = \mathit{DelPoint}(w)$, there uniquely exists a character $a \in \Sigma$ such that $xa \in \mathit{Substr}(w)$ and $\overset{w}{\overrightarrow{xa}} = w$. Let y be the string such that $xy = w$ with $y \in \Sigma^+$, where the first character of y is a. Let s be an arbitrary element in $[x]^R_w$. Since $x \in \mathit{Prefix}(w)$, $\overset{w}{\overleftarrow{x}} = x$. Thus $s \in \mathit{Suffix}(x)$, which implies $sy \in \mathit{Suffix}(w)$. □

For the case that $\mathit{DelPoint}(w) \neq \mathit{LastNode}(w)$, we have the following.

Lemma 9 *Let $w \in \Sigma^+$, $x = \mathit{DelPoint}(w)$, and $z = \mathit{LastNode}(w)$. Assume $x \neq z$. Let s be any string in $[\overset{w}{\overset{\Longrightarrow}{z}}]^R_w$. Then there uniquely exist strings $t, u \in \Sigma^+$ such that $st \in \mathit{Suffix}(x)$ and $stu \in \mathit{Suffix}(w)$.*

Proof. Since $\overset{w}{\overset{\Longrightarrow}{z}} = z$, $\overset{w}{\overrightarrow{z}} = z$. By the assumption that $z \neq x$, we have $z \in \mathit{Prefix}(x)$. Since $x = \mathit{DelPoint}(w)$, there uniquely exists a character $a \in \Sigma$ such that $\overset{w}{\overrightarrow{za}} = x$. Thus there is a unique string $t \in \Sigma^+$ such that $zt = x$. Since $z \in \mathit{Prefix}(w)$, $z = \overset{w}{\overleftarrow{z}}$. Therefore, for any string $s \in [z]^R_w$ it holds that $st \in \mathit{Suffix}(x)$. Moreover, there uniquely exists a character $b \in \Sigma$ such that $\overset{w}{\overrightarrow{xb}} = w$. Let $u \in \Sigma^+$ be the string satisfying $\overset{w}{\overrightarrow{xb}} = xu$. Now we have $ztu = w$, and for any $s \in [z]^R_w$, it holds that $stu \in \mathit{Suffix}(w)$. □

Lemma 10 *Let $w \in \Sigma^+$, $x = \mathit{DelPoint}(w)$, and $z = \mathit{LastNode}(w)$. Assume $x \neq z$. Then $x = \mathit{LRS}(w)$.*

Proof. Since $x \neq z$, $\overset{w}{\overset{\Longrightarrow}{x}} \neq \overset{w}{\overrightarrow{x}}$. Hence $\overset{w}{\overrightarrow{x}} = x \in \mathit{Suffix}(w)$ by Proposition 3. It is not difficult to show that x occurs in w just twice. Let $y = ax$ with $a \in \Sigma$, such that $y \in \mathit{Suffix}(w)$. Assume, for a contradiction, $y = \mathit{LRS}(w)$. On the assumption, y appears in w at least twice. If $y \notin \mathit{Prefix}(w)$, y must also occur in w as neither a prefix nor a suffix of w. It turns out that x appears three times in w: a contradiction. If $y \in \mathit{Prefix}(w)$, x is of the form a^ℓ. Then $y = \mathit{DelPoint}(w)$, which contradicts the assumption that $x = \mathit{DelPoint}(w)$. Consequently, $x = \mathit{LRS}(w)$. □

According to the above three lemmas, we obtain the following theorem.

Theorem 12 *Let $w \in \Sigma^+$, $x = \mathit{DelPoint}(w)$, and $z = \mathit{LastNode}(w)$. Let $k = |[\overset{w}{\overset{\Longrightarrow}{z}}]^R_w|$. Suppose $u_1, u_2, \ldots, u_k$ be the suffixes of w arranged in decreasing order of their length, where $u_1 = w$.*

1. *When $x = z$: Let $xy = w$. Assume that the edge $([\overset{w}{\overrightarrow{x}}]^R_w, y, [\overset{w}{\overrightarrow{w}}]^R_w)$ is deleted from CDAWG$'(w)$.*
2. *When $x \neq z$: Let $zt = x$ and $ztu = w$. Assume that the edge $([\overset{w}{\overrightarrow{z}}]^R_w, tu, [\overset{w}{\overrightarrow{w}}]^R_w)$ of CDAWG$'(w)$ is shortened into the edge $([\overset{w}{\overrightarrow{z}}]^R_w, t, [\overset{w}{\overrightarrow{x}}]^R_w)$.*

In both cases, the suffixes $u_1, \ldots, u_k$ are removed from the CDAWG.

What the above theorem implies is that after deleting or shortening the last edge in the backbone of *CDAWG*$'(w)$, the leftmost position of a sliding window "skips" k characters at once. Let *DelSize*$(w) = k$. The next question is the exact upper bound of *DelSize*(w). Fortunately, we achieve a reasonable result such that *DelSize*(w) is at most about half of $|w|$. A more precise evaluation will be performed in Section 6.2.4.

Can't we delete only the leftmost character of w in (amortized) constant time? We strongly believe the answer is "No". The reason is as follows. Let $|w| = n$ where $w \in \Sigma^*$. Let $u_1, u_2, \ldots, u_{n+1}$ be all the suffixes of w arranged in decreasing order of their length. It will be proven in Chapter 12 that the total number of nodes necessary to keep *CDAWG*$'(u_i)$ for every $1 \leq i \leq n+1$ is $\Theta(n^2)$, even if we minimize the CDAWGs so to share as many nodes and edges as possible. Therefore, the amortized time complexity to delete the leftmost character of w would be proportional to n.

6.2.2 Maintaining the Structure of CDAWG

Suppose the last edge of the backbone of *CDAWG*$'(w)$ is deleted or shortened right now. Let $k =$ *DelSize*(w). Let $u = w[k+1 : n]$ where $n = |w|$. We sometimes need to modify the structure of the current graph, so that it exactly becomes *CDAWG*$'(u)$. The *out-degree* of a node v is denoted by *OutDeg*(v). Let $x =$ *DelPoint*(w) of *CDAWG*$'(w)$.

Firstly, we consider when *OutDeg*$([\overset{w}{\overrightarrow{x}}]^R_w) \geq 3$ in the first cast of Theorem 12. In this case, $\overset{u}{\overrightarrow{x}} = x$ and *OutDeg*$([\overset{u}{\overrightarrow{x}}]^R_u) \geq 2$. It does not contradict Definition 13, and thus no more maintenance is required. An example of the case is shown in Figure 6.1.

Secondly, we consider when *OutDeg*$([\overset{w}{\overrightarrow{x}}]^R_w) = 2$ in the first case of Theorem 12. Let $([\overset{w}{\overrightarrow{r}}]^R_w, s, [\overset{w}{\overrightarrow{x}}]^R_w)$ be an arbitrary in-coming edge of the node $[\overset{w}{\overrightarrow{x}}]^R_w$ in *CDAWG*$'(w)$. Assume $\overset{w}{\overrightarrow{r}} = r$, that is, $rs \in$ *Suffix*(x). Let $([\overset{w}{\overrightarrow{x}}]^R_w, t, [\overset{w}{\overrightarrow{w}}]^R_w)$ be the edge which is to be the sole

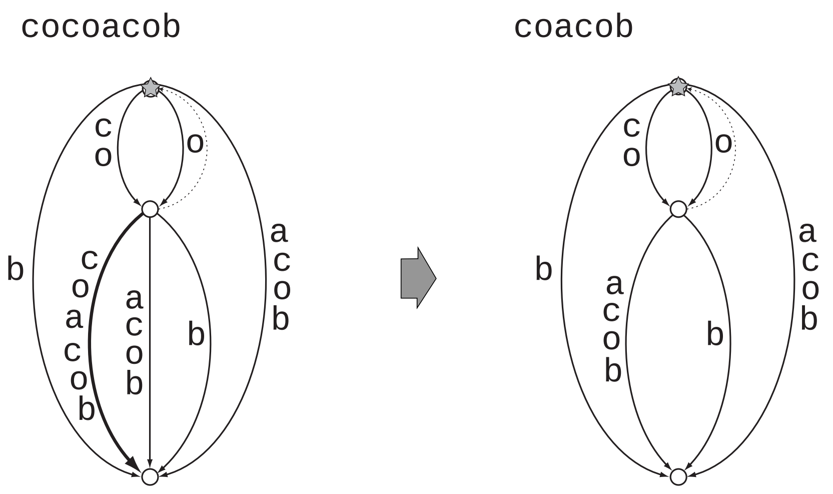


Figure 6.1: On the left, *CDAWG*′(cocoacob) is shown. The thick edge is to be deleted. The resulting structure is *CDAWG*′(coacob), shown on the right. The gray star indicates the active point for each.

remaining out-going edge of the node $[\overset{u}{\overrightarrow{x}}]^R_u$ after the deletion. Notice that, however, $\overset{u}{\overrightarrow{x}} = u$. Thus the edge $([\overset{w}{\overrightarrow{r}}]^R_w, s, [\overset{w}{\overrightarrow{x}}]^R_w)$ is modified to $([\overset{u}{\overrightarrow{r}}]^R_u, st, [\overset{u}{\overrightarrow{u}}]^R_u)$. The total time required in the operations is proportional to the number of in-coming edges of the node $[\overset{w}{\overrightarrow{x}}]^R_w$ in $\mathit{CDAWG}'(w)$. It is bounded by $\mathit{DelSize}(w)$.

Moreover, we might need a maintenance of the active point. Let $v = \mathit{LRS}(w)$. Supposing that $v \in \mathit{Prefix}(xt)$, v is represented on the edge $([\overset{w}{\overrightarrow{x}}]^R_w, t, [\overset{w}{\overrightarrow{w}}]^R_w)$ in $\mathit{CDAWG}'(w)$. The active point is actually referred to as the pair $([\overset{w}{\overrightarrow{x}}]^R_w, p)$, where $p \in \mathit{Prefix}(t)$ and $xp = v$. The reference pair is modified to $([\overset{u}{\overrightarrow{r}}]^R_u, sp)$ in $\mathit{CDAWG}'(u)$. Note that $\overset{u}{\overleftarrow{r}} \cdot sp = v$. An example of the case is shown in Figure 6.2.

Thirdly, we consider the second case in Theorem 12. In this case, the last edge in the backbone, $([\overset{w}{\overrightarrow{z}}]^R_w, tu, [\overset{w}{\overrightarrow{w}}]^R_w)$, is shortened into $([\overset{u}{\overrightarrow{z}}]^R_u, t, [\overset{u}{\overrightarrow{x}}]^R_u) = ([\overset{u}{\overrightarrow{z}}]^R_u, t, [\overset{u}{\overrightarrow{u}}]^R_u)$ in $\mathit{CDAWG}'(u)$. It implies that $x \neq \mathit{LRS}(u)$, although $x = \mathit{LRS}(w)$. The active point of $\mathit{CDAWG}'(w)$ is represented by $([\overset{w}{\overrightarrow{z}}]^R_w, t)$, since $zt = x$ (by Lemma 10). Let $\mathit{suf}([\overset{w}{\overrightarrow{z}}]^R_w) = [\overset{w}{\overrightarrow{s}}]^R_w$. Assuming $\overset{w}{\overrightarrow{s}} = s$, s is the longest string such that $s \in \mathit{Suffix}(z)$ and $s \notin [\overset{w}{\overrightarrow{z}}]^R_w$. Notice that $\mathit{LRS}(u) = st$. Hereby, the reference pair of the active point is changed to $([\overset{u}{\overrightarrow{s}}]^R_u, t)$. If $[\overset{u}{\overrightarrow{s}}]^R_u$ is the explicit parent node nearest the locus of st, we are done. If not, the reference pair is canonized to $([\overset{u}{\overrightarrow{r}}]^R_u, p)$ such that $s \in \mathit{Prefix}(\overset{u}{\overrightarrow{r}})$, $st = \overset{u}{\overrightarrow{r}} \cdot p$, and $|p|$ is minimum. An example of the case is shown in Figure 6.3.

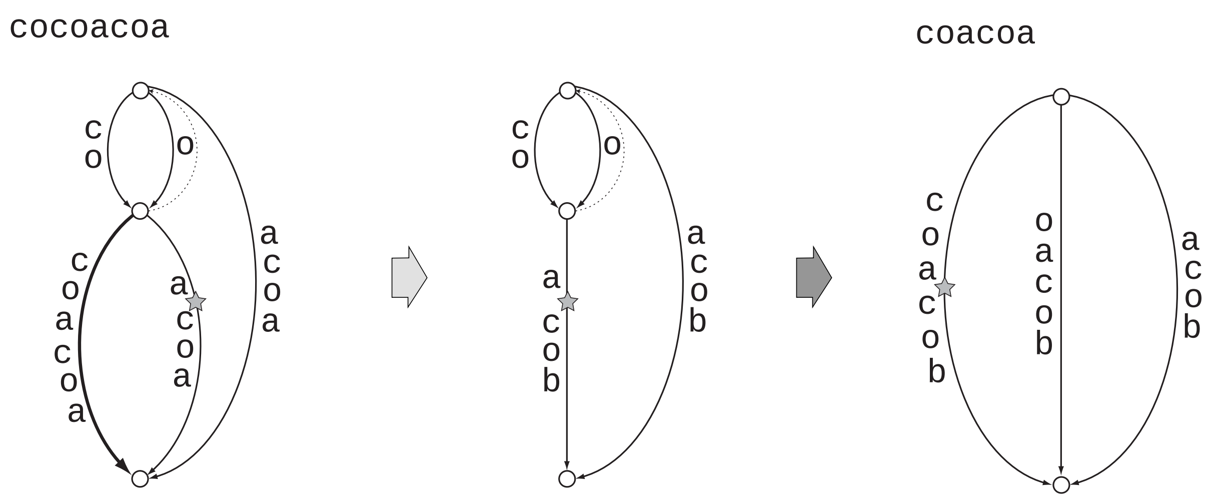


Figure 6.2: On the left, *CDAWG*′(`cocoacoa`) is shown, where the thick edge is to be deleted. The center is the intermediate structure in which the edge is deleted. After the modifications, we obtain *CDAWG*′(`coacoa`), shown on the right. The gray star indicates the active point for each.

6.2.3 Detecting *DelPoint*(u)

Suppose that after the edge deletion or shortening of *CDAWG*(w), we got *CDAWG*(u), where $u \in$ *Suffix*(w). The problem is how to locate *DelPoint*(u) in *CDAWG*(u). A naive solution is to traverse the backbone of *CDAWG*(u) from the source node. However, it takes $O(|u|)$ time, which leads to quadratic time complexity in total.

Our approach is to keep track of the *advanced point* that corresponds to the locus of $w[1 : n-1]$, where $n = |w|$. Let $x =$ *LastNode*(w) and $xy = w$, that is, $([\overset{w}{\overrightarrow{x}}]^R_w, y, [\overset{w}{\overrightarrow{w}}]^R_w)$ is the edge for deletion or shortening. The canonical reference pair for the advanced point is $([\overset{w}{\overrightarrow{x}}]^R_w, t)$, where $t \in$ *Prefix*(y) and $\overset{w}{\overrightarrow{x}} \cdot t = w[1 : n-1]$. We move to node $[\overset{w}{\overrightarrow{s}}]^R_w = suf([\overset{w}{\overrightarrow{x}}]^R_w)$. Suppose *CDAWG*$(w)$ has already been converted to *CDAWG*(u). Assume $[\overset{u}{\overrightarrow{s}}]^R_u = [\overset{w}{\overrightarrow{s}}]^R_w$. Since $\overset{u}{\overrightarrow{s}} \cdot t = u[1 : m-1]$ where $m = |u|$, $([\overset{u}{\overrightarrow{s}}]^R_u, t)$ is a reference pair of the next advanced point, and then it is canonized. Let $([\overset{u}{\overrightarrow{s'}}]^R_u, t')$ be the canonical reference pair of the advanced point. Then $\overset{u}{\overrightarrow{s'}} =$ *LastNode*(u). If *LastNode*$(u) \notin$ *Prefix*$($*LRS*$(u))$, that is, if the active point is not on the longest out-going edge from the node $[\overset{u}{\overrightarrow{s'}}]^R_u$, *DelPoint*$(u) =$ *LastNode*(u). Otherwise, *DelPoint*$(u) =$ *LRS*(u). In the case that $[\overset{u}{\overrightarrow{s}}]^R_u \neq [\overset{w}{\overrightarrow{s}}]^R_w$, we perform the same procedure from its closest parent node (see

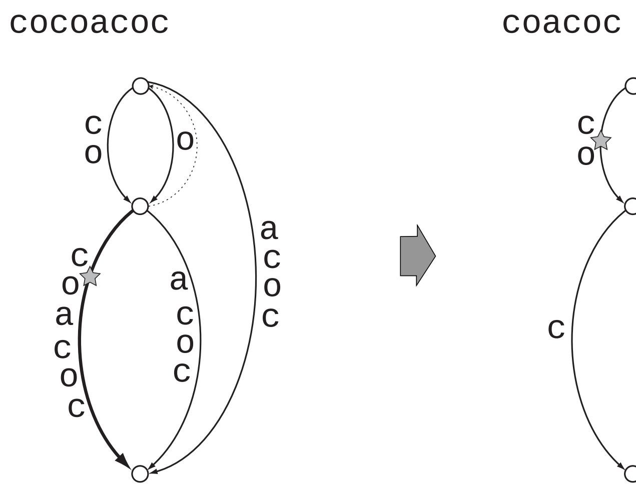


Figure 6.3: On the left, *CDAWG*′(cocoacoc) is shown. The thick edge is to be shortened. The resulting structure is *CDAWG*′(coacoc), shown on the right. The gray star indicates the active point for each.

Figure 6.2). If Σ is fixed, the cost of canonizing the reference pair is only proportional to the number of nodes included in the path. The amortized number of such nodes is constant.

6.2.4 On Buffer Size

The main theorem of this section shows an exact estimation of the upperbound of $\mathit{DelSize}(w)$. For an alphabet Σ and an integer n, we define $\mathit{MaxDel}_\Sigma(n) = \max\{\mathit{DelSize}(w) \mid w \in \Sigma^*, |w| = n\}$.

Theorem 13 *If* $|\Sigma| \geq 3$, $\mathit{MaxDel}_\Sigma(n) = \lceil \frac{n}{2} \rceil - 1$.

By this theorem, edge deletion or edge shortening can shrink the window size upto the half of the original size. Therefore, in order to keep the window size at least M, a buffer of size $2M + 1$ is necessary and sufficient.

We will prove the above theorem in the sequel.

Lemma 11 *Let* $w \in \Sigma^*$. *For any string* $x \in \mathit{Substr}(w)$, *let* $\mathit{suf}([\overset{w}{\overrightarrow{x}}]^R_w) = [\overset{w}{\overrightarrow{s}}]^R_w$. *Then* $|[\overset{w}{\overrightarrow{x}}]^R_w| = |\overset{w}{\overrightarrow{x}}| - |\overset{w}{\overrightarrow{s}}|$.

Proof. $|[\overset{w}{\overrightarrow{x}}]^R_w| = |\mathit{Suffix}(\overset{w}{\overrightarrow{x}})| - |\mathit{Suffix}(\overset{w}{\overrightarrow{s}})| = (|\overset{w}{\overrightarrow{x}}| + 1) - (|\overset{w}{\overrightarrow{s}}| + 1) = |\overset{w}{\overrightarrow{x}}| - |\overset{w}{\overrightarrow{s}}|$. $\square$

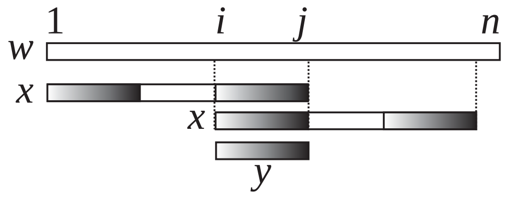


Figure 6.4: The case $j > \frac{n}{2}$. x occurs at least twice in w, and the overlap y is in fact both a prefix and suffix of x.

Lemma 12 *Let $w \in \Sigma^*$ and $n = |w|$. For any $x \in Prefix(w) - \{w\}$ with $\overset{w}{\Longrightarrow x} = x$, $|[\overset{w}{\Longrightarrow x}]^R_w| \le \lceil \frac{n}{2} \rceil - 1$.*

Proof. Let $j = |x| = |\overset{w}{\Longrightarrow x}|$. Let $suf([\overset{w}{\Longrightarrow x}]^R_w) = [\overset{w}{\Longrightarrow s}]^R_w$. We have the following three cases.

(1) When $j < \frac{n}{2}$: Since j is an integer, $j \le \lceil \frac{n}{2} \rceil - 1$, and $|[\overset{w}{\Longrightarrow x}]^R_w| = |\overset{w}{\Longrightarrow x}| - |\overset{w}{\Longrightarrow s}| \le \lceil \frac{n}{2} \rceil - 1$ by Lemma 11.

(2) When $j > \frac{n}{2}$: (See Figure 6.4.) The equivalences $x = w[1 : j]$ and $\overset{w}{\Longrightarrow x} = x$ imply that $x = w[i : i+j-1]$ for some $i \ge 2$ and $i+j-1 \le n$. Then $i-j \le n-2j+1 < 1$, that is, $i \le j$. Let $y = w[i : j]$. Its length is $|y| = j - i + 1 \ge 1$, and $y = x[i : j] \in Suffix(x)$. Since $\overset{w}{\Longrightarrow x} = x$ and $y \in Suffix(x)$, $\overset{w}{\Longrightarrow y} = y$. On the other hand, $y = w[i : j] = x[1 : j-i+1] = w[1 : j-i+1] \in Prefix(w)$, which implies $\overset{w}{\Longleftarrow y} = y$. Thus y is the longest element of $[\overset{w}{\Longrightarrow y}]^R_w$. Since $|x| > |y|$, $x \notin [\overset{w}{\Longrightarrow y}]^R_w$. Therefore $|\overset{w}{\Longrightarrow s}| \ge |y|$, which yields $|[\overset{w}{\Longrightarrow x}]^R_w| = |\overset{w}{\Longrightarrow x}| - |\overset{w}{\Longrightarrow s}| \le |x| - |y| = j - (j - i + 1) = i - 1 \le n - j < n - \frac{n}{2} = \frac{n}{2}$. Thus $|[\overset{w}{\Longrightarrow x}]^R_w| \le \lceil \frac{n}{2} \rceil - 1$.

(3) When $j = \frac{n}{2}$: Since $\overset{w}{\Longrightarrow x} = x$, x occurs in w at least twice. If $x = w[i : i + j - 1]$ for some i with $2 \le i \le j$, we can show the inequality holds in the same way as (2). Otherwise, $x = w[j + 1 : 2j] = w[j + 1 : n] = w[1 : j]$, that is $w = xx$. Then $\overset{w}{\Longrightarrow x} = w \ne x$, which does not satisfy the precondition of the lemma.

In any cases, we have got the result. □

We are ready to prove the upperbound of $MaxDel_\Sigma(n)$.

Lemma 13 $MaxDel_\Sigma(n) \le \lceil \frac{n}{2} \rceil - 1$ *for any Σ and any $n \ge 3$.*

Proof. Let $x = \mathit{DelPoint}(w)$ and $z = \mathit{LastNode}(w)$. First we consider the case $x = z$. Since $\overset{w}{\overrightarrow{x}} = x$ and $x \in \mathit{Prefix}(w) - \{w\}$, $\mathit{DelSize}(w) = |[\overset{w}{\overrightarrow{z}}]^R_w| = |[\overset{w}{\overrightarrow{x}}]^R_w| \leq \lceil \frac{n}{2} \rceil - 1$ by Lemma 12.

We now assume $x \neq z$. Then $z \in \mathit{Prefix}(x)$, and $x = \mathit{DelPoint}(w)$ implies that $x \in \mathit{Prefix}(w) - \{w\}$, which yields $z \in \mathit{Prefix}(w) - \{w\}$. Thus by Lemma 12, $\mathit{DelSize}(w) = |[\overset{w}{\overrightarrow{z}}]^R_w| \leq \lceil \frac{n}{2} \rceil - 1$. □

On the other hand, the lowerbound is given by the following lemma.

Lemma 14 *If* $|\Sigma| \geq 3$, $\mathit{MaxDel}_\Sigma(n) \geq \lceil \frac{n}{2} \rceil - 1$ *for any* $n \geq 1$.

Proof. For each $1 \leq n \leq 4$, the inequality trivially holds since $\lceil \frac{n}{2} \rceil - 1 \leq 1$. Let $\mathtt{a}, \mathtt{b}, \mathtt{c}$ be distinct symbols in Σ. For each odd $n \geq 5$, let $w_n = \mathtt{a}^k\mathtt{ba}^k\mathtt{bc}$, where $k = \frac{n-3}{2}$. Remark that $\mathit{DelPoint}(w_n) = \mathtt{a}^k\mathtt{b}$. Let $x = \mathtt{a}^k\mathtt{b}$. We can see that any suffix of x except ε belongs to $[\overset{w}{\overrightarrow{x}}]^R_w$, so that $\mathit{suf}(x) = \varepsilon$. Thus $\mathit{DelSize}(w_n) = |x| - |\varepsilon| = |\mathtt{a}^k\mathtt{b}| - 0 = k+1 = \frac{n-1}{2} = \lceil \frac{n}{2} \rceil - 1$, since n is odd. For each even $n \geq 6$, let $w'_n = \mathtt{a}^{k-1}\mathtt{ba}^k\mathtt{bc}$, where $k = \frac{n}{2} - 1$, and we can verify that $\mathit{DelSize}(w'_n) = \lceil \frac{n}{2} \rceil - 1$ similarly. □

Consequently, Theorem 13 is proved by Lemma 13 and Lemma 14. We note that for a binary alphabet $\Sigma = \{\mathtt{a}, \mathtt{b}\}$, two series of the strings $\mathtt{a}^k\mathtt{ba}^k\mathtt{bab}$ and $\mathtt{a}^{k-1}\mathtt{ba}^k\mathtt{bab}$ give the lowerbound $\mathit{MaxDel}_\Sigma(n) \geq \lceil \frac{n-3}{2} \rceil$.

On the on-line algorithm of Chapter 4, each node $[\overset{w}{\overrightarrow{x}}]^R_w$ of $\mathit{CDAWG}'(w)$ stores the value of $|\overset{w}{\overrightarrow{x}}|$. By Lemma 11, it is guaranteed that we can calculate $\mathit{DelSize}(w)$ in constant time with no additional information.

6.2.5 Keeping Edge Labels Valid

As mentioned previously, an edge label is actually represented by a pair of integers indicating its beginning and ending positions in input string w, respectively. We must ensure that no edge label becomes "out of date" after the window slides, e.g., that no integer refers to a position outside the sliding window. In case of a suffix tree, when a new edge is created, we can guarantee the above regulation by traversing from the leaf node toward the root node while updating all edge labels encountered. However, this would yield quadratic time complexity in the aggregate. Larsson [49, 50] utilized *credit issuing*, an update-number-restriction technique, originally proposed in [22], which takes in total $O(|w|)$ time and space. In the following, we introduce an extended credit issuing technique for CDAWGs. Our basic strategy is to show that we can handle the credit issuing as well as in case of suffix trees.

We assign each internal node s of $\mathit{CDAWG}'(w)$ a binary counter called *credit*, denoted by $\mathit{Cred}(s)$. This credit counter is initially set to zero when s is created. When a node s receives a credit, we update the labels of in-coming edges of s. Then, if $\mathit{Cred}(s) = 0$, we set it to one, and stop. If $\mathit{Cred}(s) = 1$, after setting it to zero, we let the node s issue a credit to its parent nodes.

When s is newly created, $\mathit{Cred}(s) = 0$. The creation of the new node s implies that a new edge is to be inserted from s to the sink node. When the new edge is created leading to the sink node, the sink node *issues* a credit to the parent node s. Assume the new edge is labeled by pair (i, j) where i, j are some integers with $i < j$. Let ℓ be the length of the label of the in-coming edge of s. After s received a credit from the sink node, we reset its in-coming edge label to $(i - \ell, i - 1)$. Remember the edge redirection happening in the construction of a CDAWG (see Section 4.3.1). If some edge is actually redirected to node s, its label is updated as well. Note that we need not change the value of $\mathit{Cred}(s)$ again.

Suppose a node r has right now received a credit from one of its child nodes. Assume $\mathit{Cred}(r)$ is currently one. We need to update all in-coming edge labels of r. We store a list in r to maintain its in-coming edges arranged in the order of the length of the path they correspond to. The maintenence of the list is an easy matter, since the on-line algorithm of Chapter 4 inserts edges to r in such order. Let t be an arbitrary parent node of r. Let k be the number of the in-coming edges of r connected from t. One might wonder that r must issue k credits to t, but there is the following time-efficient method. In case k is even, $\mathit{Cred}(t)$ need not to be changed because it is a *binary* counter. Contrarily, in case k is odd, we always change the value of $\mathit{Cred}(t)$. If $\mathit{Cred}(t)$ was one, we also have to update the in-coming edge of t. To do it, we focus on the *shortest* in-coming edge of r connected from t, which is in turn the shortest out-going edge of t leading to r. In updating the in-coming edges of t, we should utilize the label of the shortest edge, since the label corresponds to the possibly newest occurrence of the substrings represented in node t. We continue updating edge labels by traversing the reversed graph rooted at r in width-first manner while issuing credits.

Recall the node separation in constructing a CDAWG (see Section 4.3.1). Assume a node r has right now been created owing to the separation of a node s. The subgraph rooted at r is currently the same as the one rooted at s, since r was created as a clone of s. Thus we simply set $\mathit{Cred}(r) = \mathit{Cred}(s)$.

Now consider a node u to be deleted, corresponding to the second case of Section 6.2.2. It might have received a credit from its newest child node (that is not deleted), which has not been issued to its parent node yet. Therefore, when a node u is scheduled for deletion and $\mathit{Cred}(u) = 1$, node u issues credits to its parent nodes. However, this complicates the update of edge labels: several waiting credits may aggregate, causing nodes upper in the CDAWG to receive a credit *older* than the one it has already received from its another child node. Therefor, before updating an edge label, we compare its previous value against the one associated with the received credit, and refer to the newer one. As well as the case of edge insertion mentioned in the above paragraph, we traverse the reversed graph rooted at u in width-first fashion to update edge labels. In the worst case, the updating cost is proportional to the number of paths from the source node to node u. Nevertheless, it is bounded by $\mathit{DelSize}(w)$.

By analogous arguments to [22, 49, 50], we can establish the following lemma.

Lemma 15 *All edge labels of a CDAWG can be kept valid in a sliding window, in linear time and space with respect to the length of an input string.*

As a conclusion of Section 6.2, we finally obtain the following.

Theorem 14 *Let $w \in \Sigma^*$ and M be the window size. The proposed algorithm runs in $O(|w|)$ time using $O(M)$ space.*

Chapter 7

On-Line Construction of Symmetric CDAWGs

As seen in literature [77, 55, 9, 10, 73, 18] and the previous chapters, *suffix links* are often used, and essential, for time-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 $STrie(w)$ exactly form the edges of $STrie(w^{\text{rev}})$ [24]. DAWGs also have a similar property, that is, the suffix links of the $DAWG(w)$ compose $STree(w^{\text{rev}})$ [12]. However, this duality is damaged in case of suffix trees. Namely, the suffix links of $STree(w)$ do not always form a structure supporting indexes of w^{rev}. Still, the set of suffix links of $STree(w)$ is a subset of the set of edges of $DAWG(w^{\text{rev}})$ [16].

In order to obtain the complete duality on suffix trees, Stoye [66, 67] introduced *affix trees*. Affix trees are the modification of suffix trees so that the suffix links of $ATree(w)$ form $ATree(w^{\text{rev}})$ (see Figure 7.3). Stoye could not prove his on-line algorithm for constructing affix trees runs in linear time, but Maaß [51] later on gave an improved version of the algorithm which runs in linear time. Meanwhile, Blumer et al. [10] showed that the nodes of a CDAWG are invariant under reversal: there is a one-to-one correspondence between the nodes of $CDAWG(w)$ and those of$CDAWG(w^{\text{rev}})$. The structure with those two kinds of edges is called the *symmetric compact directed acyclic word graph* (*SCDAWG*) of w (see Figure 7.2, right).

Ukkonen [73] gave two intuitive and excellent on-line algorithms for constructing $STrie(w)$ and $STree'(w)$, respectively, as recalled in Chapter 4. Since the suffix links of $STrie(w)$ are equal to the edges of $STrie(w^{\text{rev}})$, it turns out that $STrie(w)$ and $STrie(w^{\text{rev}})$

sharing the same nodes can be simultaneously built on-line, scanning w from left to right. Also, since the algorithm of [9] constructs DAWGs on-line, it can be regarded as an algorithm which builds $\mathit{DAWG}(w)$ and $\mathit{STree}(w^{\mathrm{rev}})$ at the same time, in on-line (left to right) fashion. Moreover, the fact is that the algorithm by Weiner [77] that constructs suffix trees becomes more interesting when considered as an on-line algorithm. His algorithm builds $\mathit{STree}(w)$ by appending the suffixes of w to the current suffix tree in increasing order. In other words, his algorithm builds $\mathit{STree}(w)$ on-line, *right to left*. In addition to that, his algorithm can be modified so as to create the edges of $\mathit{DAWG}(w^{\mathrm{rev}})$ at the same time [18]. It implies that his algorithm also simultaneously constructs $\mathit{DAWG}(w)$ together with $\mathit{STree}(w^{\mathrm{rev}})$ on-line, left to right.

In this chapter, we first give an algorithm that simultaneously builds $\mathit{STree}'(w)$ with $\mathit{DAWG}(w^{\mathrm{rev}})$ on-line, left to right. This algorithm constructs $\mathit{STree}(w)$ in the same way as Ukkonen's algorithm, while computing the *shortest extension links* (sext links) that correspond to the edges of $\mathit{DAWG}(w^{\mathrm{rev}})$ at the same time. Moreover, we show an algorithm that *directly* constructs $\mathit{SCDAWG}(w)$ *on-line*, left to right. It builds $\mathit{CDAWG}'(w)$ similarly to the algorithm we introduced in Chapter 4, and computes the sext links that are actually the edges of $\mathit{CDAWG}'(w^{\mathrm{rev}})$.

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

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

holds for any string $w \in \Sigma^*$. This is because, intuitively, the set of nodes of $\mathit{SCDAWG}(w)$ is the *intersection* of those of $\mathit{STrie}(w)$ and $\mathit{STrie}(w^{\mathrm{rev}})$, while the set of nodes in $\mathit{ATree}(w)$ is the *union* of them. Therefore, SCDAWGs considerably save memory space, compared to affix trees. Moreover, not only SCDAWGs are 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 [68].

The result in this chapter was originally published in [37].

7.1 Bidirectional Index Structures

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

Giegerich and Kurtz [24] observed that $\mathit{STrie}(w)$ and $\mathit{STrie}(w^{\mathrm{rev}})$ are dual in the sense that they share the same nodes. We refer this bidirectional index structure as "$\mathit{STrie}(w)$ with $\mathit{STrie}(w^{\mathrm{rev}})$". A formal definition follows.

Definition 22 $\mathit{STrie}(w)$ with $\mathit{STrie}(w^{\mathrm{rev}})$ is the bidirectional tree $(V, E_{L\to R}, E_{R\to L})$ such that

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

It is obvious that there is a trivial one-to-one correspondence between $E_{R\to L}$ and F for the suffix links of $\mathit{STrie}(w)$ in Definition 8.

The duality of $\mathit{STree}(w)$ and $\mathit{DAWG}(w^{\mathrm{rev}})$, which was pointed out in [12, 16], is shown in Definition 23.

Definition 23 $\mathit{STree}(w)$ with $\mathit{DAWG}(w^{\mathrm{rev}})$ is the bidirectional dag $(V, E_{L\to R}, E_{R\to L})$ such that

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

Let $V' = \{[x]^{L}_{w} \mid x \in \mathit{Substr}(w)\}$. It is easy to see that there is a trivial one-to-one correspondence between the node set V of Definition 23 and V'. Using this correspondence, we can identify $E_{R\to L}$ of Definition 23 with

$$\begin{array}{cl} & \{([x]^{L}_{w}, a, [ax]^{L}_{w}) \mid x, ax \in \mathit{Substr}(w) \text{ and } a \in \Sigma\} \\ = & \{([y]^{R}_{w^{\mathrm{rev}}}, a, [ya]^{R}_{w^{\mathrm{rev}}}) \mid y, ya \in \mathit{Substr}(w^{\mathrm{rev}}) \text{ and } a \in \Sigma\}, \end{array}$$

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

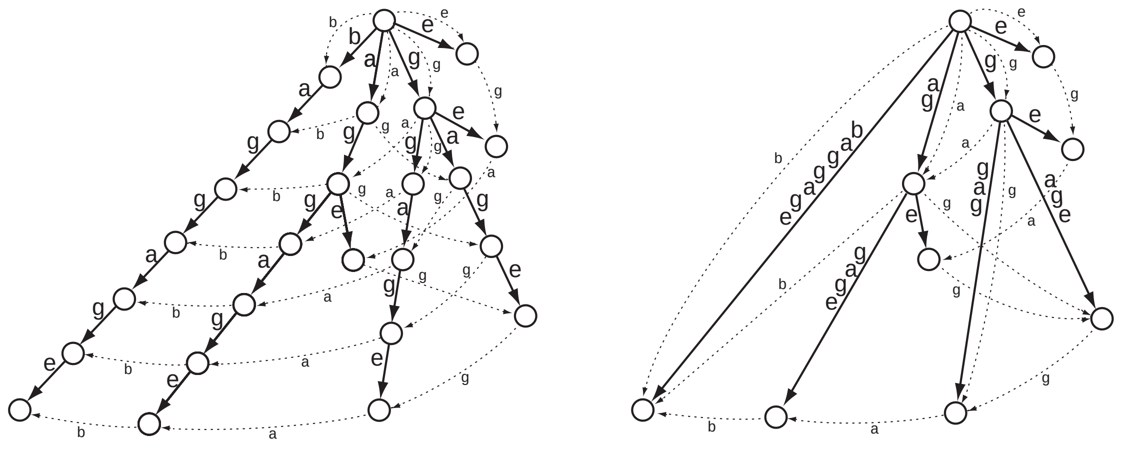


Figure 7.1: *STrie*(w) with *STrie*(w^{rev}) on the left, and *STree*(w) with *DAWG*(w^{rev}) on the right, where $w =$ `baggage`. The thick solid lines represent the edges of *STrie*(w) and *STree*(w), while the thin break lines do the ones of *STrie*(w^{rev}) and *DAWG*(w^{rev}), respectively. Since `baggage` ends with a unique character `e`, the end-marker \$ is omitted.

The edges $E_{R \to L}$ of Definition 23 are the so-called *shortest extension links* (*sext links*) of *STree*(w) [16]. Moreover, a part of the reversed sext links are known as suffix links. Recalling the definition, the suffix links are the set

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

The reversal of the suffix links with labels are called *reversed suffix link*, defined by

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

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 Figure 7.1 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 13 can be transformed as follows:

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

In Definition 24, we show the definition of the *symmetric CDAWG* (*SCDAWG*) of a string w, denoted by *SCDAWG*(w).

Definition 24 *SCDAWG*(w) is the bidirectional dag $(V, E_{L\to R}, E_{R\to L})$ such that

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

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

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

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 *DAWG*(w) with *STree*(w^{rev}), and *SCDAWG*(w) in Figure 7.2, where $w =$ `baggage`.

Another symmetric bidirectional index structure, called *affix trees*, was introduced by Stoye [66]. *ATree*(w) and *ATree*(w^{rev}) for $w =$ `baggage` are shown in Figure 7.3 without a formal definition for comparison. Intuitively, the set of the nodes in *SCDAWG*(w) is the *intersection* of those in *STree*(w) and *STree*(w^{rev}), while the set of the nodes in *ATree*(w) is the *union* of them.

7.2 On-Line Construction of *STree*(w) with *DAWG*(w^{rev})

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

7.2.1 "*STree*(w) with *DAWG*(w^{rev})" Redefined

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

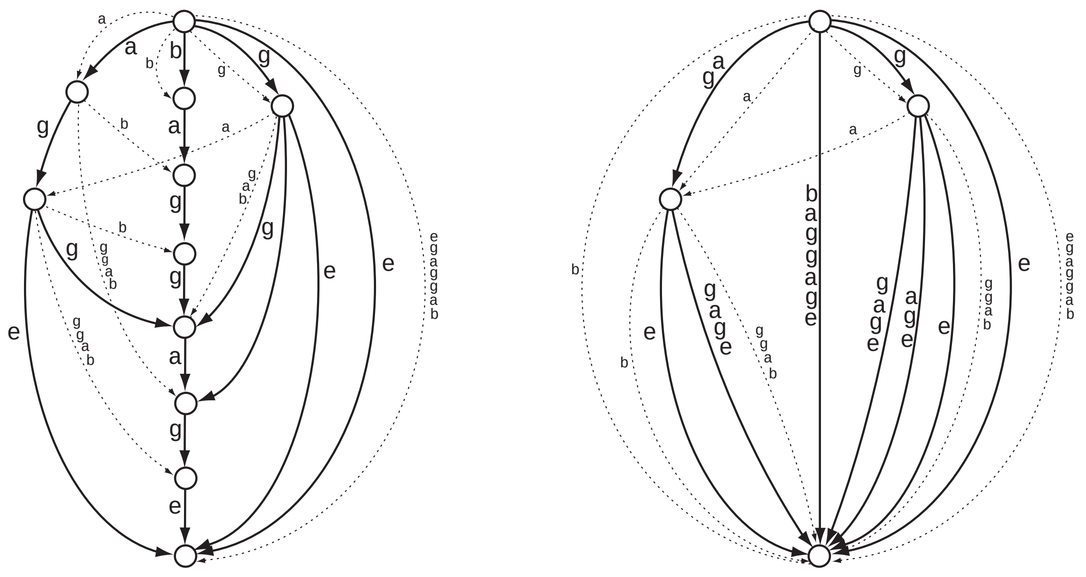


Figure 7.2: *DAWG*(w) with *STree*(w^{rev}) on the left, and *SCDAWG*(w) on the right, for string $w = \texttt{baggage}$.

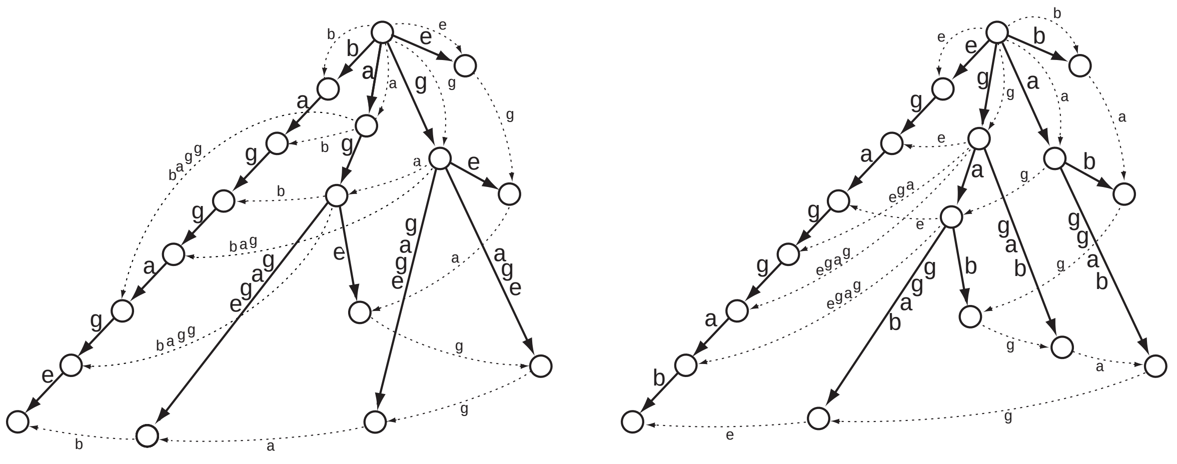


Figure 7.3: *ATree*(w) on the left and *ATree*(w^{rev}) on the right, where $w = \texttt{baggage}$.

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

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

Since $\overset{w\$}{\overrightarrow{x}} = \overset{w\$}{\overrightarrow{x}}$, this structure is identical to that defined in Definition 23 for input string $w\$$.

7.2.2 Main Idea of the Algorithm

As for $STree'(w)$ for a string $w \in \Sigma^*$, our algorithm creates it in entirely the same way as Ukkonen's algorithm (see Section 4.2 or [73]). Every time a new node is created during the construction of $STree'(w)$, the sext links of the new node, which correspond to certain edges of $DAWG(w^{rev})$, 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 $STree'(w)$, 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 $STree'(w)$ can be quadratic due to a series of updating the open edges. However, recall the fact that label α of an edge of $STree'(w)$ is usually implemented with a pair of integers (i, j) such that $\alpha = w[i : j]$. Furthermore, note that the second value of the label of any open edge in $STree'(w[1 : h])$ is h for $1 \leq h \leq n$. Therefore, if we implement the second value with a global variable, we can update all the open edges in constant time with 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 11 *Suppose that in $STree'(w)$ the reversed suffix link of a leaf node x, which is labeled a, points to a node y. Then node y is also a leaf node in $STree'(w)$.*

Proof. From the definition the reversed suffix link of node x is a triple $(\overset{w}{\overrightarrow{x}}, a, \overset{w}{\overrightarrow{ax}})$ 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.

7.2.3 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 Figure 7.5 that shows each phase of the construction of $STree'(\#\texttt{abab}\$)$. The starred point in Figure 7.5 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 Figure 7.5. 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 Figure 7.4, left). Only v is the newly created node in case (b$_1$). Concrete examples

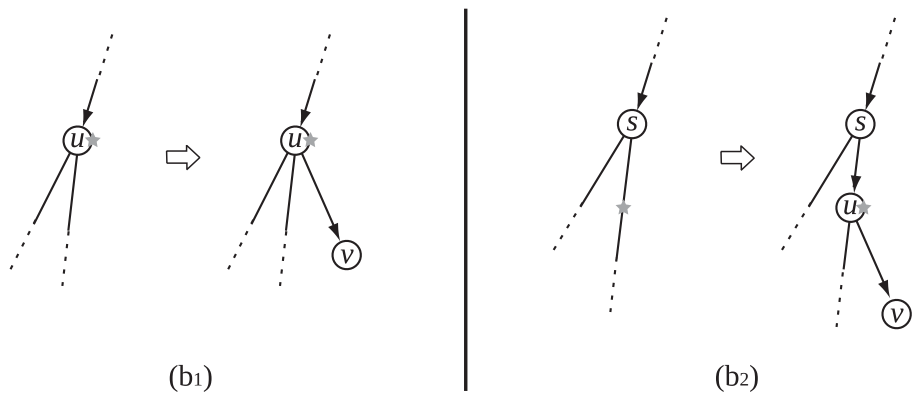


Figure 7.4: The two cases of the position of the active point, which is denoted by a gray star. Since the active point is on a node u in case (b_1) displayed on the left, only leaf node v is newly created. On the other hand, in case (b_2) on the right, internal node u is also created where the active point is at present, in the middle of an edge.

can be seen in phases #, #`a`, #`ab`, and the third step of phase #`abab`\$ in Figure 7.5. 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 Figure 7.4, right). Concrete examples of case (b_2) can bee seen at the first and second steps in phase #`abab`\$ in Figure 7.5. 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).

Sext Link of a Leaf Node

In both cases (b_1) and (b_2), it follows from Proposition 11 that the reversed suffix link of a new leaf node v points to the last created leaf node v'. Suppose v is the jth created leaf node and v' is $(j-1)$th one during the construction of $STree'(w)$, where $2 \leq j \leq |w|$. Then the reversed suffix link of node v pointing to v' is labeled by $w[j-1]$, in formula, $rsuf[v, w[j-1]] = v'$. We have the following proposition which concerns with the sext link of v.

Proposition 12 *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.*

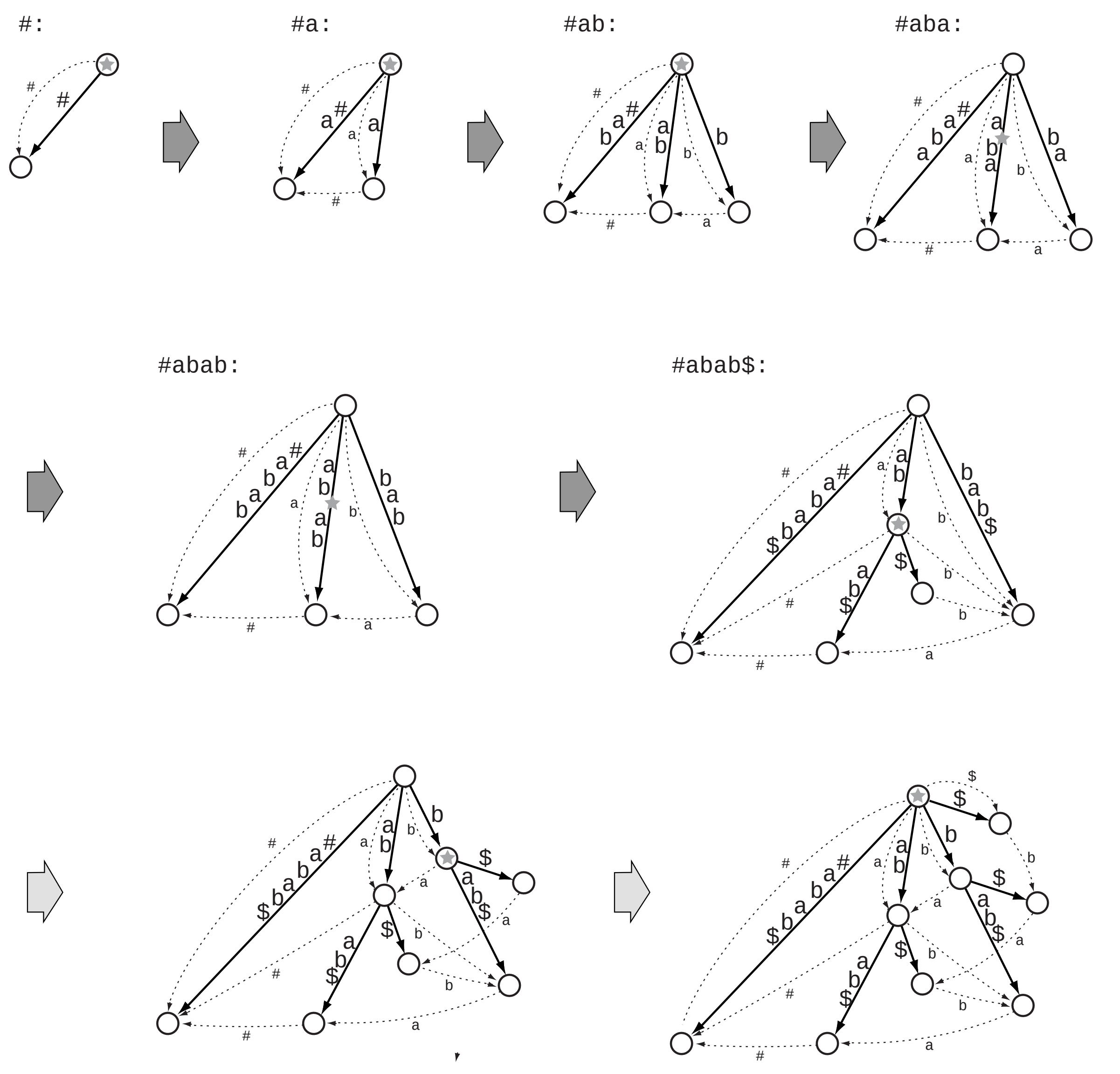


Figure 7.5: The on-line construction of $STree'$(#abab\$) with the sext links represented by the broken arrows. At the third step of phase #abab\$, the sext links form $DAWG$(\$baba#).

Proof. Since v is a leaf node, v is a substring 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 $Substr(w)$. $\square$

For example, leaf node b is created in phase #ab of Figure 7.5, 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 $\mathit{suf}[u'] = u$, then let a be the character such that $\mathit{rsuf}[u, a] = u'$. Suppose there is a node r' such that $\mathit{sext}[r, b] = r'$ with $b \neq a$. Then $\mathit{sext}[u, b]$ is set to point to r' as well, since $r' = \overset{\xRightarrow{w[1:i]}}{bu}$ in this case (remember the definition of sext links). For instance, at the first step of phase $\#\mathtt{abab}\$$ in Figure 7.5, $u = \mathtt{ab}$, $r = \mathtt{abab}\$$ and $r' = \mathit{sext}[\mathtt{abab}\$, \#] = \#\mathtt{abab}\$$. Since $\mathit{rsuf}[\mathtt{ab}, \#]$ is undefined, we define $\mathit{sext}[\mathtt{ab}, \#] = \#\mathtt{abab}\$$. If $b = a$, then $\mathit{sext}[u, b]$ stays pointing to node u', because obviously $u' = \overset{\xRightarrow{w[1:i]}}{bu} = \overset{\xRightarrow{w[1:i]}}{au}$. For example, at the second step of phase $\#\mathtt{abab}\$$ in Figure 7.5, $\mathit{sext}[\mathtt{bab}\$, \mathtt{a}] = \mathtt{abab}\$$. However, since $\mathit{rsuf}[\mathtt{b}, \mathtt{a}] = \mathtt{ab}$, $\mathit{sext}[\mathtt{b}, \mathtt{a}] = \mathtt{ab}$. As previously remarked in the reason (ii) in Section 7.2.2, 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 16 *When an internal node u is newly created in phase $w[1:i]$ during the construction of $\mathit{STree}'(w)$ 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 $suf[u] = u'$. Namely, node v currently represents $w[j : i+1]$ and node p corresponds to $w[j+1 : i+1]$. That is why $sext[p, w[j]] = v$. If the active point again stops on a node until the algorithm faces case (b), the sext link of the node whose label is $w[j]$ is created and set to point to the leaf node v as well. A concrete example is shown in Figure 7.6.

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 Figure 7.5. 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, $\overset{w[1:j]}{\overrightarrow{\overrightarrow{cp}}} = r$. If $|u| > |p|$, then the sext link of p is modified so as to point to u ($sext[p, c] = u$), because $\overset{w[1:i]}{\overrightarrow{\overrightarrow{cp}}} = u$. A concrete example can be seen between phase #`abab` and phase #`abab`\$ in Figure 7.5. $sext[\varepsilon, \mathtt{a}] = \mathtt{abab}$ 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 $\overset{w[1:i]}{\overrightarrow{\overrightarrow{cp}}} = r$ in this case. Similar discussion holds for the sext links pointing to node v, another child of node u.

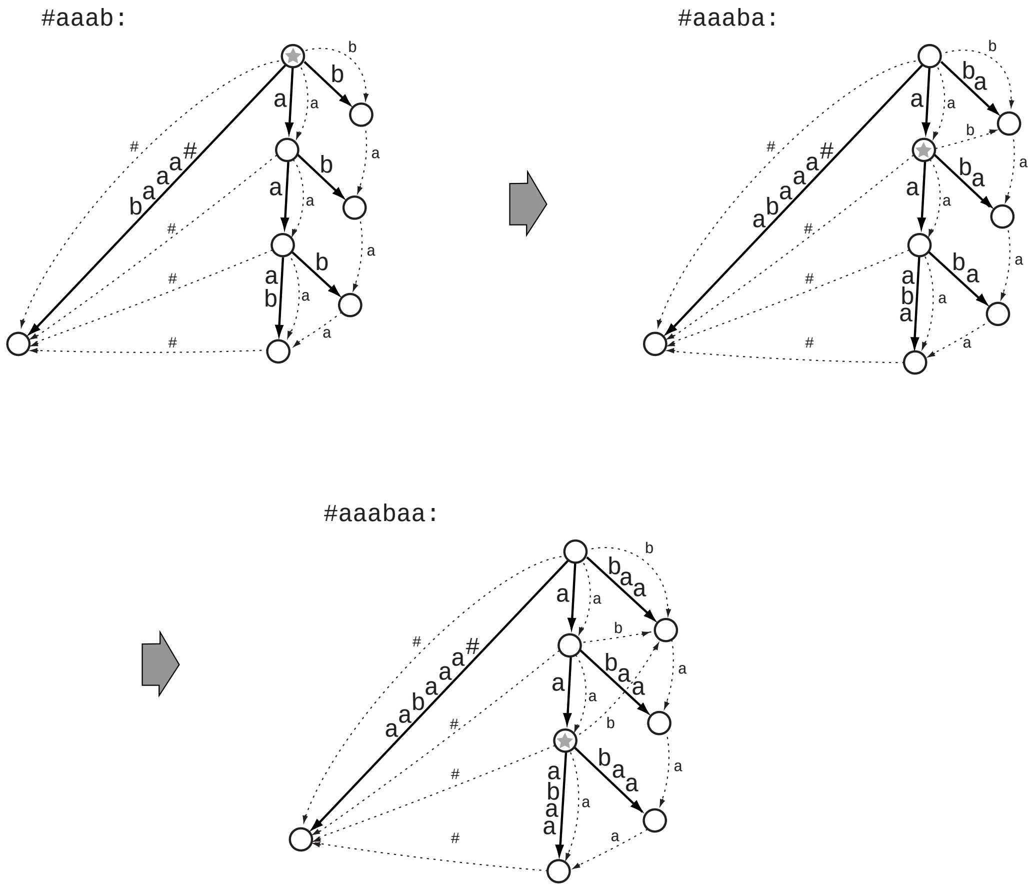


Figure 7.6: *STree*′(#`aaab`) with the sext links is shown on the left. Node `b` is the last created leaf node in that phase. Scanning a new character `a`, the active point moves to node `a`, as seen in the center figure *STree*′(#`aaaba`). Then, *sext*[`a`, `b`] is set to point to the last created leaf node `ba`. Also in the right figure representing *STree*′(#`aaabaa`), *sext*[`aa`, `b`] = `baa`, because the active point has arrived at node `aa`.

7.2.4 Correctness and Complexity of the Algorithm

The algorithm is summarized as Figure 7.7. If we compute the sext links of the nodes in "*STree*′(w) with sext links" according to the algorithm, we have the following:

Theorem 15 *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 [73] 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

```
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 7.7: The algorithm to construct "*STree*$'(w)$ with sext links". *check_end_point*, *extension*, *canonize*, and *split_edge* are identical to those used in Chapter 4.

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. □

7.3 On-Line Construction of SCDAWGs

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 [39], 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 [39] is based on the Ukkonen suffix tree construction algorithm. This implies, if we add the functions "*redirect*" and "*separate_node*" in [39] to the pseudo-code of the algorithm in Section 7.2, 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 [26] as an exercise. Furthermore, the CDAWG construction algorithm in [39] 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 7.2 should be changed when constructing CDAWGs, by using examples. If again turning our attention to the pseudo-code of Figure 7.7, the 7th 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.

7.3.1 Sext Link Corresponding to a Newly Created Edge

A sequence of snapshots on the on-line construction of $SCDAWG'(\#\texttt{abab}\$)$ is shown in Figure 7.8. 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 Figure 7.5, 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 $(w[1:j])^{rev}$.

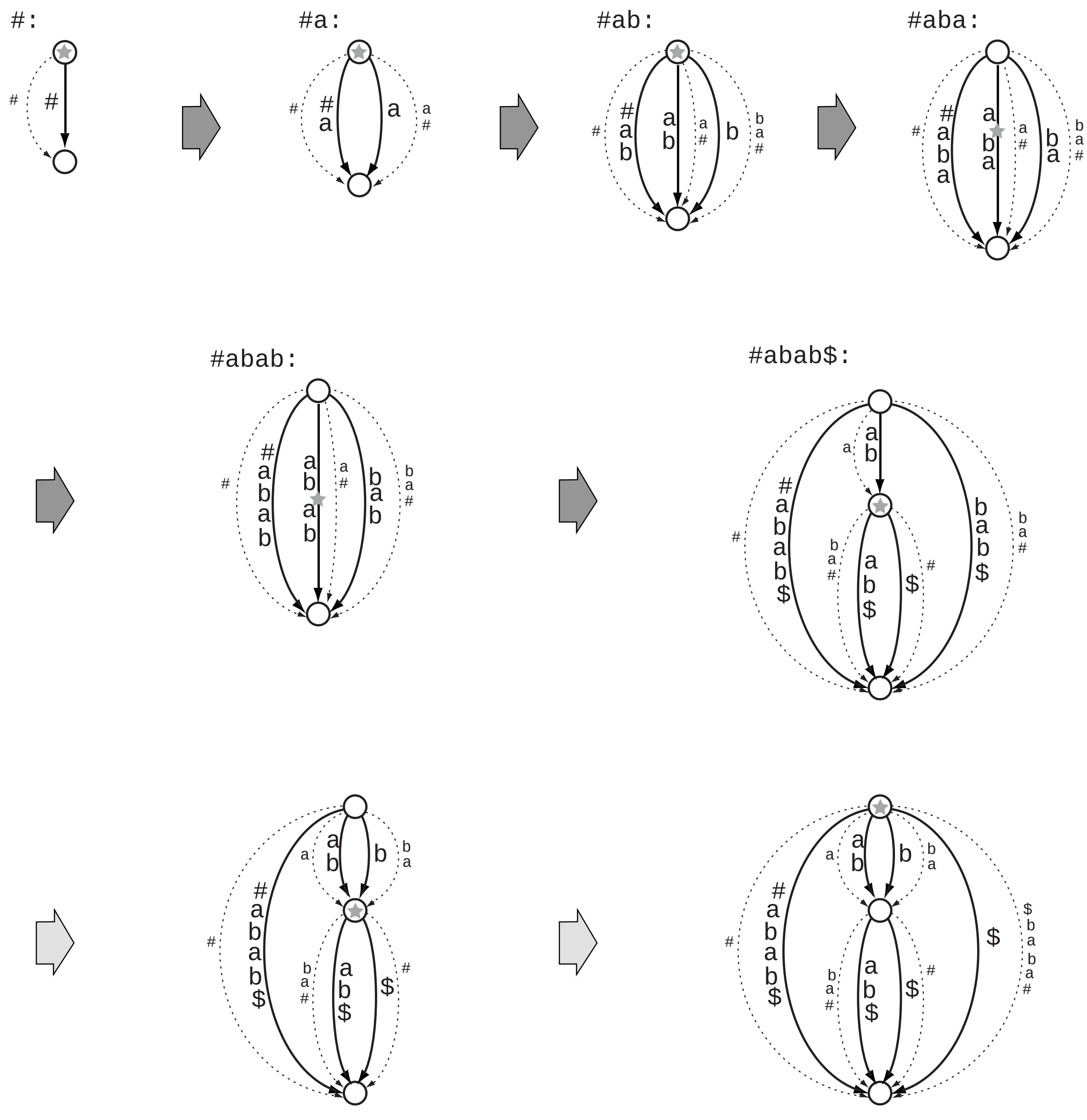


Figure 7.8: The on-line construction of $SCDAWG'(w)$, where $w = \#\texttt{abab}\$$. The solid arrows represent the edges of $CDAWG'(w)$, whereas the broken arrows represent the sext links of the nodes of $CDAWG'(w)$, that are equivalent to the edges of $CDAWG(w^{rev})$.

7.3.2 Change of Sext Links

See phases #`abab` and #`abab`$ in Figure 7.8. 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

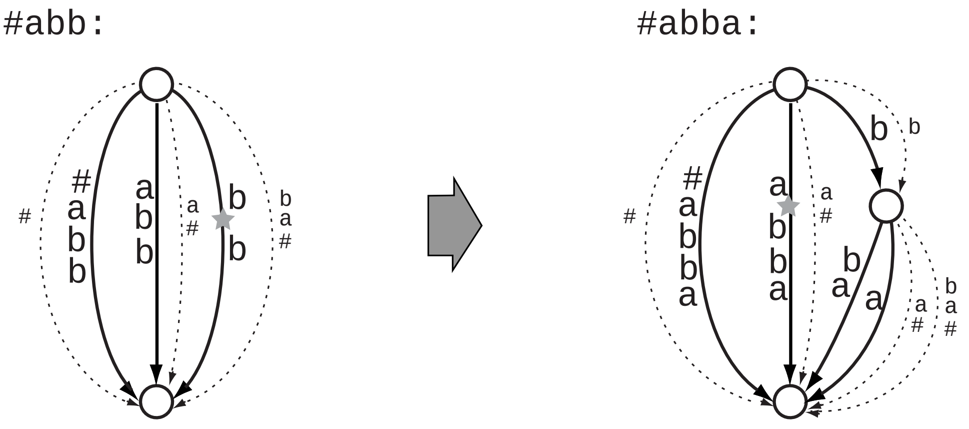


Figure 7.9: *CDAWG*′(#abb) and *CDAWG*′(#abba) with their sext links. The sext link *sext*[ε, ba#] is cut into two sext links *sext*[ε, b] and *sext*[b, a#].

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 $\overset{w[1:6]}{\overrightarrow{\#ab}} = \#\mathtt{abab}\$$ in this time, where $w[1:6] = \#\mathtt{abab}\$$.

What if a sext link, whose label is of length more than 1, is cut? See Figure 7.9 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 *sext*[ε, ba#] = #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 *CDAWG*′(#abb) in Figure 7.9, *sext*[ε, #] = #abb,

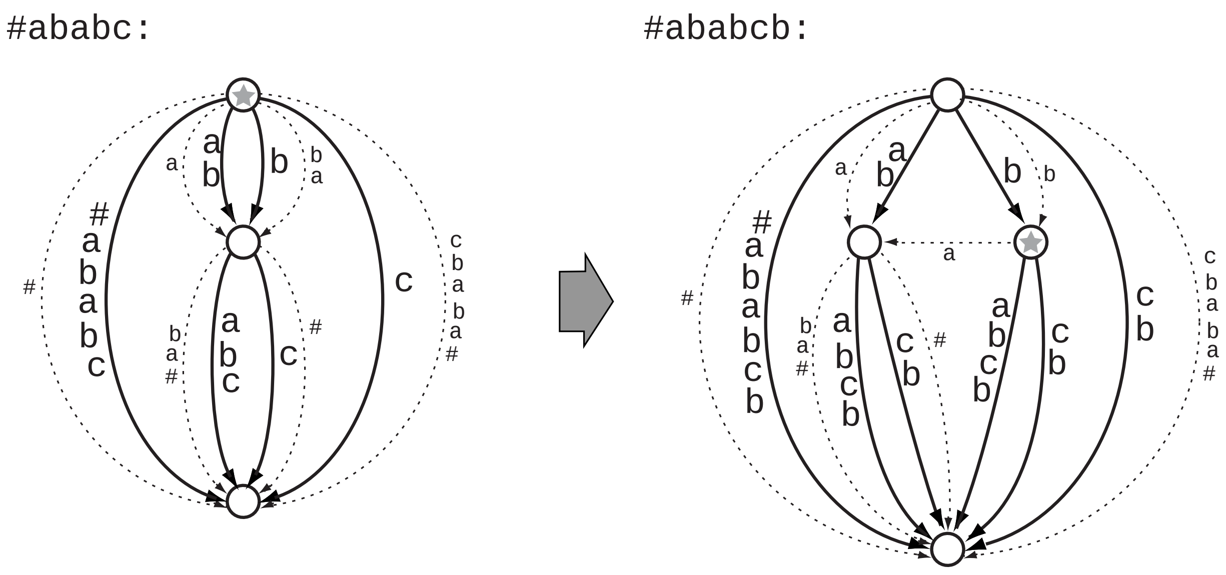


Figure 7.10: *CDAWG*′(#`ababc`) and *CDAWG*′(#`ababcb`) with their sext links.

sext[ε, `a`#] = `abb`, *sext*[ε, `ba`#] = `bb`.

As seen in phase #`abab`$ of Figure 7.8, the edge labeled with `bab`$ is *merged* into the node `ab` and its label is modified to `b`. According to it, *sext*[ε, `ba`#] = `bab`$ becomes *sext*[ε, `ba`] = `b`. The character `a` at the tail of label `ba` of the sext link corresponds to the label of the sext link from node `b` to `ab` in *STree*(#`abab`$) in Figure 7.5.

Figure 7.10 displays a node *separation* that can happen during the construction of CDAWGs. In Figure 7.10, 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 *CDAWG*′(#`ababcb`). Then, *sext*[ε, `ba`] = `b` in *CDAWG*′(#`ababc`) is cut into two, one of which is *sext*[`b`, `a`] = `ab` and the other *sext*[ε, `b`] = `b`.

7.3.3 Implementation of Substrings Represented in a Node

As is mentioned above, a sext link in *CDAWG*′(w) is set to point to a certain substring of w represented in a node. However, if we actually implement all of such strings naively, the space requirement can be quadratic. Therefore, we implement them with integers referring to the positions in the input string w. Suppose that the representative of a node p is α in *CDAWG*′($w[1:i]$) for $1 \leq i \leq |w|$. Then, node p has integers j and k ($1 \leq j \leq k \leq i$) such that $\alpha = w[j:k]$ where j represents the beginning position of the left most occurrence of α in $w[1:i]$. In addition to it, each edge entering node p has an integer representing the entrance order to node p. See the left figure in Figure 7.10, $\mathit{CDAWG}'(\#\mathtt{ababc})$. 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 $\mathit{CDAWG}'(\#\mathtt{ababc})$, the edge labeled `abc` entering to the final node represents two substrings `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 16 *For any string $w \in \Sigma^*$, SCDAWG for w can be constructed on-line in linear time and space with respect to $|w|$.*

Chapter 8

Bidirectional Construction of Suffix Trees

Among the index structures we have mentioned so far, suffix trees are for sure most widely-known and extensively-studied [16, 26], perhaps because there are a 'myriad' of applications for them [4]. Construction of suffix trees has been studied in various contexts: Weiner [77] invented the first algorithm that constructs suffix trees in linear time; McCreight [55] proposed a more space-economical algorithm; Chen and Seiferas [12] showed an efficient modification of Weiner's; Ukkonen [73] introduced an on-line algorithm to construct suffix trees, which Giegerich and Kurtz [24] regarded as "the most elegant"; Farach [21] considered optimal construction of suffix trees with large alphabets; Breslauer [11] gave a linear-time algorithm for building the suffix tree of a given trie that stores a set of strings; Shibuya [62] considered construction of the suffix tree for a trie with a large alphabet; Kurtz [47] pursued reducing space requirement of suffix trees.

In this paper we explore *bidirectional construction* of suffix trees. Namely, the algorithm we propose allows us to update $STree'(w)$ to $STree'(xwy)$ with any strings $x, y, w \in \Sigma^*$. We also show that our algorithm runs in linear time and space with respect to the length of a given string.

Bidirectional construction of suffix trees was first considered by Stoye [66, 67]. His strategy was to modify the definition and structure of suffix trees so that they become more "adequate" in terms of bidirectional construction, and the resulting modified structure was named *affix trees.* His original algorithm to construct affix trees does not perform in linear time, unfortunately, but Maaß [51] later on improved the algorithm so as to run in linear time. Another good feature of affix trees is that the affix tree of any string w

also supports the indices of w^{rev}, the reversal of w. On the other hand, it is a well-known and beautiful property that the suffix tree of any string w and the DAWG of w^{rev} can share the same nodes, as well. We will show that the combination of this work and the algorithm given in Chapter 7 enables us to construct and update both structures in a bidirectional manner.

The size of affix trees is, of course, linear. However, for any string w the number of nodes in the suffix tree for w is less than or equal to that of the affix tree. Namely, this work contributes to reducing space requirements necessary for bidirectional construction of a data structure that supports dual indices of a given string.

This result primarily appeared in [34].

8.1 Bidirectional Construction of Suffix Trees

8.1.1 Right Extension

Assume that we have $STree'(w)$ with some $w \in \Sigma^*$. Now we consider updating it into $STree'(wa)$ with $a \in \Sigma$, by inserting the suffixes of wa into $STree'(w)$. By Theorem 6 it is clear that, for any $a \in \Sigma$ and $w \in \Sigma^*$, $STree'(w)$ can be updated to $STree'(wa)$ in amortized constant time. See Figure 4.2 that shows the construction of $STree'($`cocoa`$)$ with right extension.

Here we only recall essence of Ukkonen's algorithm together with some supporting lemmas and propositions. Recall Definition 19 for $LRS(w)$ with $w \in \Sigma^*$.

Lemma 17 *Let $a \in \Sigma$ and $w \in \Sigma^*$. Let $y = LRS(w)$. For any string $x \in Suffix(w) - Suffix(y)$, $\overset{wa}{\overrightarrow{x}} = \overset{w}{\overrightarrow{x}} \cdot a$.*

Proof. Since $y = LRS(w)$, any string $x \in Suffix(w) - Suffix(y)$ appears only once in w as a suffix of w, and is therefore $\overset{w}{\overrightarrow{x}} = x$. Also, x is followed only by a in wa, and thus $\overset{wa}{\overrightarrow{x}} = xa$. □

The above lemma implies that a leaf node of $STree'(w)$ is also a leaf node in $STree'(wa)$. Thus we need no explicit maintenance for leaf nodes. Namely, we can insert all strings of $Suffix(w) - Suffix(y)$ into $STree'(w)$ *automatically* (for more detail, see Section 4.2). Now we can focus only on the suffixes of $LRS(wa)$.

Proposition 13 *Let $a \in \Sigma$ and $w \in \Sigma^*$. Let $y = LRS(w)$ and $z = LRS(wa)$. For any string $x \in Suffix(y) - Suffix(z)a^{-1}$, $\overset{wa}{\overrightarrow{x}} = x$.*

Proof. Since $x \in Suffix(y)$, x has appeared at least twice in w. Because $x \notin Suffix(z)a^{-1}$, $xa \notin Suffix(z)$. These facts imply the existence of $b \in \Sigma$ such that $xb \in Substr(w)$ and $b \neq a$. Consequently, we have $\overset{wa}{\overrightarrow{x}} = x$. □

The above proposition implies that if $x \in Suffix(y)$, $\overset{w}{\overrightarrow{x}} \neq x$ ($\overset{w}{\overrightarrow{x}}$ is implicit in $STree'(w)$), and $\overset{wa}{\overrightarrow{x}} = x$ ($\overset{wa}{\overrightarrow{x}}$ will be explicit in $STree'(wa)$), a new explicit node $\overset{wa}{\overrightarrow{x}} = x$ has to be created in the update of $STree'(w)$ to $STree'(wa)$. Plus, a new leaf node $\overset{wa}{\overrightarrow{xa}} = xa$ is created with the new edge $(\overset{wa}{\overrightarrow{x}}, a, \overset{wa}{\overrightarrow{xa}})$. Owing to the eliminator symbol ξ of Definition 18, we can establish the following lemma.

Lemma 18 *Let $a \in \Sigma$ and $w \in \Sigma^*$. Let $y = LRS(w)$ and $z = LRS(wa)$. Assume $x \in Suffix(y) - Suffix(z)a^{-1}$. Suppose t is the longest string in $Prefix(x)$ such that $\overset{w}{\overrightarrow{t}} = t$. Let $x' = Suffix(x)$ with $|x'| + 1 = |x|$, and $t' = Suffix(t)$ with $|t'| + 1 = |t|$. For the string $\alpha \in \Sigma^*$ such that $t\alpha = x$, $t'\alpha = x'$.*

Notice that we can reach string x' via the suffix link of the node for t in $STree'(w)$ and along the path spelling out α from the node for t' (recall Definition 16). For instance, see the 1st and 2nd phases for `cocoa` in Figure 4.2. After creating new explicit nodes $\overset{w}{\overrightarrow{\texttt{co}}}$ and $\overset{w}{\overrightarrow{\texttt{coa}}}$, the star mark goes backward to the parent node of $\overset{w}{\overrightarrow{\texttt{co}}}$, which is the root node $\overset{w}{\overrightarrow{\varepsilon}}$. Then it moves to $\perp$ node via the suffix link of $\overset{w}{\overrightarrow{\varepsilon}}$, and goes down along edges with spelling out `co`. The star mark is now on the location for `o`, where a new explicit node will be created in the next phase. This operation is continued until the star mark reaches $LRS($`cocoa`$)$. Ukkonen [73] proved that the amortized cost of this operation is constant, on the assumption that every edge label α of $STree'(w)$ is actually implemented by a pair (i, j) of integers such that the substring of w beginning at position i and ending at position j is α.

8.1.2 Left Extension

Weiner [77] proposed an algorithm to construct $STree(aw)$ by updating $STree(w)$ with $a \in \Sigma$ in amortized constant time. On the other hand, what we treat here is the conversion of $STree'(w)$ into $STree'(aw)$. From here on we delve in what happens to $STree'(w)$ when updated to $STree'(aw)$.

Lemma 19 *Let $a \in \Sigma$ and $w \in \Sigma^*$. For any string $x \in \mathit{Substr}(w) - \mathit{Prefix}(aw)$, $\overset{w}{\overrightarrow{x}} = \overset{aw}{\overrightarrow{x}}$.*

Proof. Since $x \notin \mathit{Prefix}(aw)$, there is no new occurrence of x in aw. Thus we have $[x]'^L_w = [x]'^L_{aw}$. □

The above lemma ensures that any implicit node of $\mathit{STree}'(w)$ does not become explicit in $\mathit{STree}'(aw)$ if it is not associated with any prefix of aw.

Now we turn our attention to the strings in $\mathit{Prefix}(aw)$. Basically, we have to insert prefixes of aw into $\mathit{STree}'(w)$ in order to obtain $\mathit{STree}'(aw)$. However, no strings in set $\mathit{Substr}(w) \cap \mathit{Prefix}(aw)$ need to be newly inserted since they are already in $\mathit{STree}'(w)$.

Definition 26 Let $a \in \Sigma$ and $w \in \Sigma^*$. The *longest repeated prefix* (*LRP*) of aw is the longest element of set $\mathit{Substr}(w) \cap \mathit{Prefix}(aw)$.

The LRP of aw is denoted by $\mathit{LRP}(aw)$. In updating $\mathit{STree}'(w)$ to $\mathit{STree}'(aw)$, we have to insert all prefixes of aw that are longer than $\mathit{LRP}(aw)$, into $\mathit{STree}'(w)$.

Lemma 20 *Let $a \in \Sigma$ and $w \in \Sigma^*$. For any $x = \mathit{Prefix}(aw) - \mathit{Substr}(w)$, $\overset{aw}{\overrightarrow{x}} = aw$.*

Proof. String x is a prefix of aw which is longer than $\mathit{LRP}(aw)$. This implies that there is no occurrence of x in w. Therefore x appears in aw exactly once as a prefix of aw, meaning there exists a unique character that follows x in aw. Hence $\overset{aw}{\overrightarrow{x}} = aw$. □

The above lemma means that, simply by adding the new leaf node $\overset{aw}{\overrightarrow{aw}} = aw$ to $\mathit{STree}'(w)$, we can obtain $\mathit{STree}'(aw)$. Moreover, the in-coming edge of the leaf node $\overset{aw}{\overrightarrow{aw}}$ will be inserted from the node that corresponds to $\mathit{LRP}(aw)$. We now clarify what happens to $\mathit{LRP}(aw)$ when the new prefixes of aw are inserted to $\mathit{STree}'(w)$.

Proposition 14 *Let $a \in \Sigma$ and $w \in \Sigma^*$. Let $x = \mathit{LRP}(aw)$ and $y = \mathit{LRS}(w)$. If $x \notin \mathit{Suffix}(w) - \mathit{Suffix}(y)$, then $\overset{aw}{\overrightarrow{x}} = x$. Otherwise, $\overset{aw}{\overrightarrow{x}} = aw$.*

Proof. We first consider the case that $x \notin \mathit{Suffix}(w) - \mathit{Suffix}(y)$. Recall that x is the *longest* string in $\mathit{Substr}(w) \cap \mathit{Prefix}(aw)$. Moreover, $x \notin \mathit{Suffix}(w) - \mathit{Suffix}(y)$. Hence, there exist two characters $b, c \in \Sigma$ such that $xb, xc \in \mathit{Substr}(aw)$ and $b \neq c$. Thus we have $\overset{aw}{\overrightarrow{x}} = x$.

Now we consider the second case, $x \in \mathit{Suffix}(w) - \mathit{Suffix}(y)$. Here, x occurs only once in w as its suffix. Thus $\overset{w}{\overrightarrow{x}} = x$. On the other hand, by the definition of $\mathit{LRP}(aw)$, we obtain $x \in \mathit{Prefix}(aw) - \{aw\}$. Therefore, there uniquely exists a character $d \in \Sigma$ which follows x in aw. Hence we have $\overset{aw}{\overrightarrow{x}} = aw$. $\square$

The above proposition implies that if $\mathit{LRP}(aw)$ does not correspond to a leaf node of $\mathit{STree}'(w)$, it will be represented by an explicit node in $\mathit{STree}'(aw)$, and otherwise, it becomes implicit in $\mathit{STree}'(aw)$ (see the 3rd and 4th steps of Figure 8.3 to be shown later on). We stress that this characterizes a difference between $\mathit{STree}'(w)$ and $\mathit{STree}(w)$. More concretely, Weiner's original algorithm constructs $\mathit{STree}(aw)$ on the basis of the next proposition.

Proposition 15 *For any $a \in \Sigma$ and $w \in \Sigma^*$, if $x = \mathit{LRP}(aw)$, then $\overset{aw}{\overrightarrow{x}} = x$.*

Now the next question is how to locate $\mathit{LRP}(aw)$ in $\mathit{STree}'(w)$. Our idea is similar to Weiner's strategy for constructing $\mathit{STree}(w)$. Let y be the longest element in set $\mathit{Prefix}(w) \cup \{\xi\}$ such that $ay \in \mathit{Substr}(w)$. Then y is called the *base* of aw and denoted by $\mathit{Base}(aw)$.

Lemma 21 (Weiner [77]) *Let $a \in \Sigma$ and $w \in \Sigma^*$. If $y = \mathit{Base}(aw)$, then $ay = \mathit{LRP}(aw)$.*

Proof. Assume contrarily that y' is the string such that $ay' = \mathit{LRP}(aw)$ and $|y'| > |y|$. By the definition of $\mathit{LRP}(aw)$, we have $ay' \in \mathit{Prefix}(aw)$, which yields $y' \in \mathit{Prefix}(w)$. It, however, contradicts the precondition that $y = \mathit{Base}(aw)$ since $|y'| > |y|$. $\square$

According to the above lemma, $\mathit{Base}(aw)$ can be a clue to locating $\mathit{LRP}(aw)$ in $\mathit{STree}'(w)$.

Let z be the longest element in set $\mathit{Prefix}(w) \cup \{\xi\}$ such that $\overset{w}{\overrightarrow{az}} = az$. Then z is called the *bridge* of aw and denoted by $\mathit{Bridge}(aw)$.

Lemma 22 *Let $a \in \Sigma$ and $w \in \Sigma^*$. If $x = \mathit{LRP}(w)$, $y = \mathit{Base}(aw)$ and $z = \mathit{Bridge}(aw)$, then $y \in \mathit{Prefix}(x)$ and $z \in \mathit{Prefix}(y)$.*

Proof. By Lemma 21 we have $ay = \mathit{LRP}(aw)$. It is easy to see that $|\mathit{LRP}(aw)| \leq |\mathit{LRP}(w)| + 1$, which implies $|y| \leq |x|$. Now we obtain $y \in \mathit{Prefix}(x)$. It can be readily shown that $az \in \mathit{Prefix}(ay)$, since $ay = \mathit{LRP}(aw)$. Thus we have $z \in \mathit{Prefix}(y)$. $\square$

Let $y = \mathit{Base}(aw)$ and $z = \mathit{Bridge}(aw)$. Assume $\gamma \in \Sigma^*$ is the string satisfying $z\gamma = y$. Then, we have $az\gamma = \mathit{LRP}(aw)$ by Lemma 21 and Lemma 22.

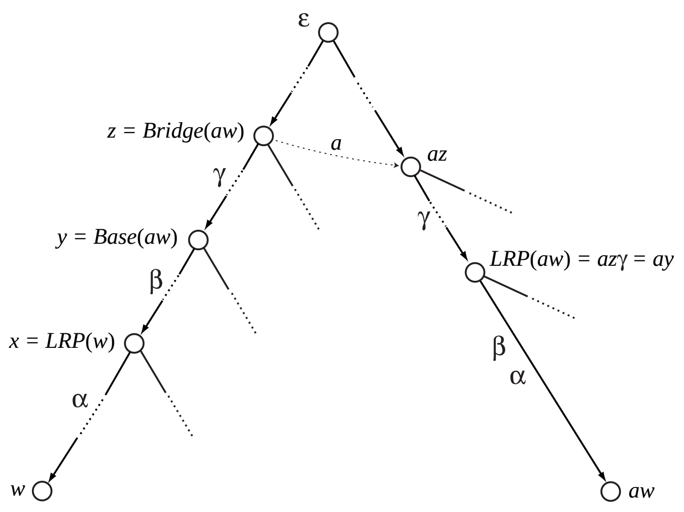


Figure 8.1: In $STree'(w)$ we start from $LRP(w)$ and go up until $Bridge(w)$. Then we move to az and go down along the path spelling out γ. We are now on the location for $LRP(aw)$, and from there we insert a new edge labeled with $\beta\alpha$. Now all prefixes of aw are inserted, we have $STree'(aw)$.

The detection of $LRP(aw)$ in $STree'(w)$ is illustrated in Figure 8.1. We start from $LRP(w)$ and then go up the path backward until encountering $Bridge(aw)$. We move to the node for az and go down the path spelling out γ, and now we are at the location for $LRP(aw)$. Finally we insert a new edge labeled with $\beta\alpha$ from the location for $LRP(aw)$ due to Lemma 20, and the resulting structure is $STree'(aw)$. The dashed arrow from $Bridge(aw) = z$ to az is the *labeled reversed suffix link* of z. The set F' of the links of $STree'(w)$ is defined as follows.

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

Observe that there is a one-to-one correspondence between F and F' for $STree'(w)$ (see Definition 9).

In order that we can find $Base(aw)$ efficiently, we maintain a table for each explicit node, as well as Weiner's algorithm. For every explicit node this table can be computed in constant time and space for any fixed alphabet. Note that, however, $Base(w)$ can sometimes be associated with an implicit node in $STree'(w)$. The following lemma shows a property of $Base(w)$ when it is implicit in $STree'(w)$.

Lemma 23 *Let $a \in \Sigma$ and $w \in \Sigma^*$. Let $y = Base(aw)$. If $y \neq \overset{w}{\overrightarrow{y}}$, then $y = LRS(w)$.*

Proof. Since $ay = \mathit{LRP}(aw)$, y appears at least twice in w. We now consider the following three cases.

1. All occurrences of y in w are followed by same character b. In this case, string yb turns out to be a prefix of w that is longer than y and appears more than once in w. It means that $y \neq \mathit{Base}(w)$, which is a contradiction.

2. There exist at least two distinct characters b, c such that $yb, yc \in \mathit{Substr}(w)$. In this case, $\overset{w}{\overrightarrow{y}} = y$, a contradiction.

3. One occurrence of y in w is followed by *no* character. This implies that y is a suffix of w.

Therefore only the third case is possible. This case, we have $y \in \mathit{Suffix}(w)$ and $y \in \mathit{Prefix}(w)$, which implies that y is the longest string satisfying the condition. Therefore, $y = \mathit{LRS}(w)$. □

Since $y = \mathit{LRS}(w)$, it is guaranteed that $\overset{w}{\overrightarrow{ay}} = ay$. We hereby regard y as *Bridge*(w) and maintain the labeled reversed suffix link of $\mathit{LRS}(w)$, which is always associated with the shortest leaf node of $\mathit{STree}'(w)$.

By a similar argument to [77], it can be established that the amortized amount of time needed for the detection of $\mathit{LRP}(aw)$ in $\mathit{STree}'(w)$ is constant, again on the assumption that every edge label is implemented by a pair of integers.

We now have the following theorem.

Theorem 17 *For any $a \in \Sigma$ and $w \in \Sigma^*$, $\mathit{STree}'(w)$ can be updated to $\mathit{STree}'(aw)$ in amortized constant time.*

Figure 8.2 shows the construction of *STree*′(`cocoa`) with left extension.

8.1.3 Mutual Influences

Here, we consider mutual influences between right extension and left extension of suffix trees. The next lemma shows what happens to $\mathit{LRP}(w)$ when $\mathit{STree}'(w)$ is updated to $\mathit{STree}'(wa)$.

Lemma 24 *Let $a \in \Sigma$ and $w \in \Sigma^*$. Assume $\mathit{LRP}(w) = \mathit{LRS}(w)$. Let $x = \mathit{LRS}(w)$. If $xa \in \mathit{Prefix}(w)$, then $\mathit{LRP}(wa) = xa$.*

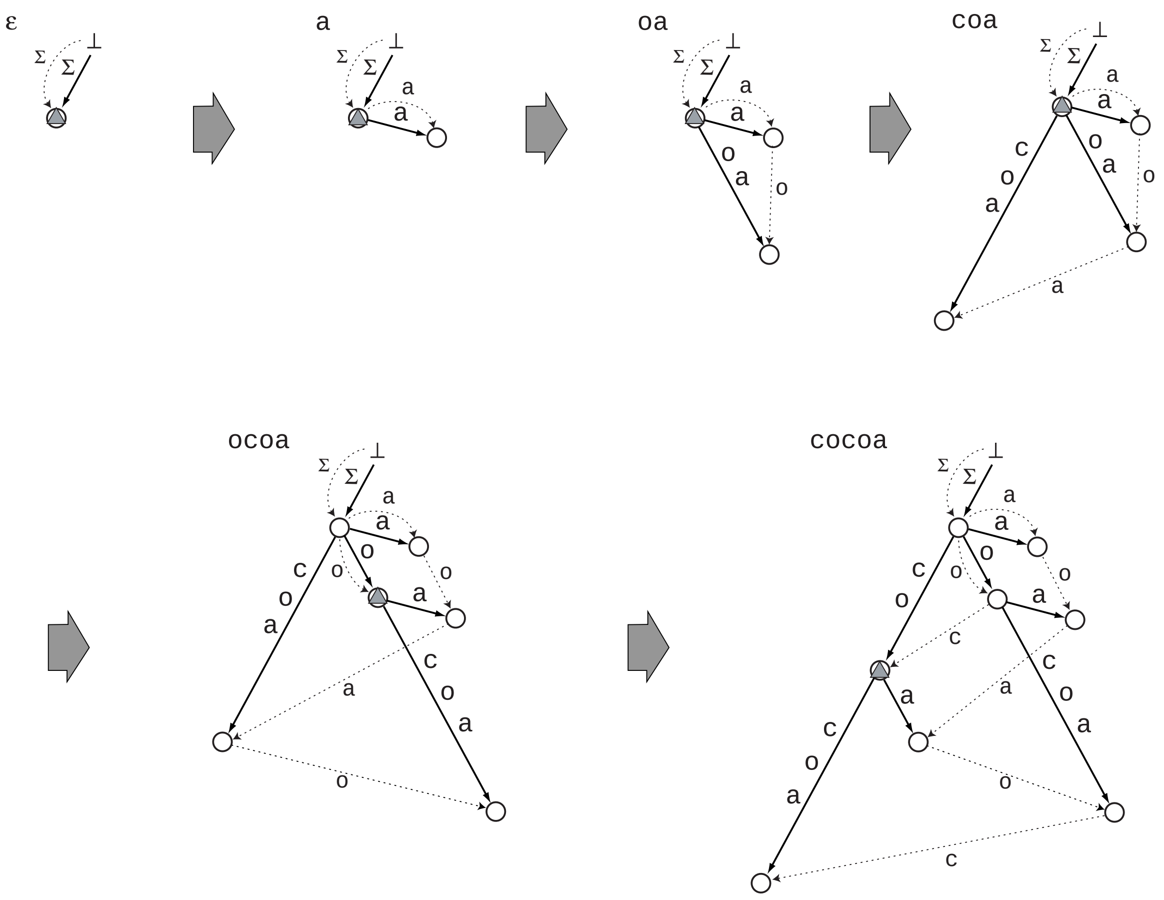


Figure 8.2: The construction of *STree*$'(w)$ with left extension, where $w =$ `cocoa`. The triangle mark represents the longest repeated prefix of each suffix tree. The $\perp$ node corresponds to the eliminator symbol ξ. The Σ symbol represents *any* character in the alphabet.

Proof. Since $xa \in \mathit{Prefix}(w)$, $\mathit{LRS}(wa) = xa$. Thus $xa = \mathit{LRP}(wa)$. □

This lemma shows when and where $\mathit{LRP}(wa)$ moves from the location of $\mathit{LRP}(w)$ according to the character a newly added to the right of w (see the 5th step in Figure 8.3). Examining the precondition, "if $xa \in \mathit{Prefix}(w)$", is feasible in $O(|\Sigma|)$ time, which is regarded as $O(1)$ if Σ is a fixed alphabet.

The following lemma stands in contrast to Lemma 24.

Lemma 25 *Let $a \in \Sigma$ and $w \in \Sigma^*$. Assume $\mathit{LRP}(w) = \mathit{LRS}(w)$. Let $x = \mathit{LRP}(w)$. If $ax \in \mathit{Suffix}(w)$, then $\mathit{LRS}(aw) = ax$.*

This lemma shows when and where $\mathit{LRS}(aw)$ moves from the location of $\mathit{LRS}(w)$ according to the character a newly added to the left of w. Examining the precondition, "if

$ax \in$ *Suffix*(w)", is also possible in $O(|\Sigma|)$ time, and moving from $LRS(w)$ to $LRS(aw)$ is possible in constant time by the use of the labeled reversed suffix link of $LRS(w)$ (see the 3rd and 4th steps of Figure 8.3).

As a result of discussion, we finally obtain the following:

Theorem 18 *For any string $w \in \Sigma^*$, STree$'(w)$ can be constructed in bidirectional manner and in $O(|w|)$ time.*

A bidirectional construction of *STree*$'(w)$ with $w =$ `ababac` is displayed in Figure 8.3.

8.2 Concluding Remarks

We introduced an algorithm for bidirectional construction of suffix trees, which performs in linear time. This is a counterpart of the algorithm of Stoye [66] for bidirectional construction of affix trees. We stress that our new algorithm requires less space than Stoye's.

An interesting fact is that the tables for finding *Base*(w) used in Weiner's algorithm [77] correspond to the edges of $DAWG(w^{\mathrm{rev}})$ [16]. This implies that our algorithm is also able to update a DAWG to the *right* direction. On the other hand, Chapter 7 presented a linear-time algorithm that constructs not only *STree*$'(w)$ but also $DAWG(w^{\mathrm{rev}})$ in a left-to-right on-line manner, which is based on the algorithm by Ukkonen [73]. This algorithm enables us to update a DAWG to the *left* direction. Therefore, the algorithm of this paper turns out to be adaptive to bidirectional construction of DAWGs.

An interesting open problem is whether or not linear-time bidirectional construction of CDAWGs is possible. It can be done in amortized constant time to convert $CDAWG(w)$ into $CDAWG(wa)$ by the use of the algorithm of Chapter 4. However, we conjecture that the conversion of $CDAWG(w)$ to $CDAWG(aw)$ would not be possible in (amortized) constant time (also see Chapter 11). Still, there might remain a possibility to construct CDAWGs in a right-to-left on-line manner. That is, a chunk of characters x (namely a string x) are at once appended to the left of the current string w, so that updating $CDAWG(w)$ to $CDAWG(xw)$ can be done in amortized constant time (this case we would not obtain $CDAWG(vw)$ for every proper suffix v of x excepting $v = \varepsilon$). However, we are at the moment unsure if such convenient selection of the length of x is really possible or not.

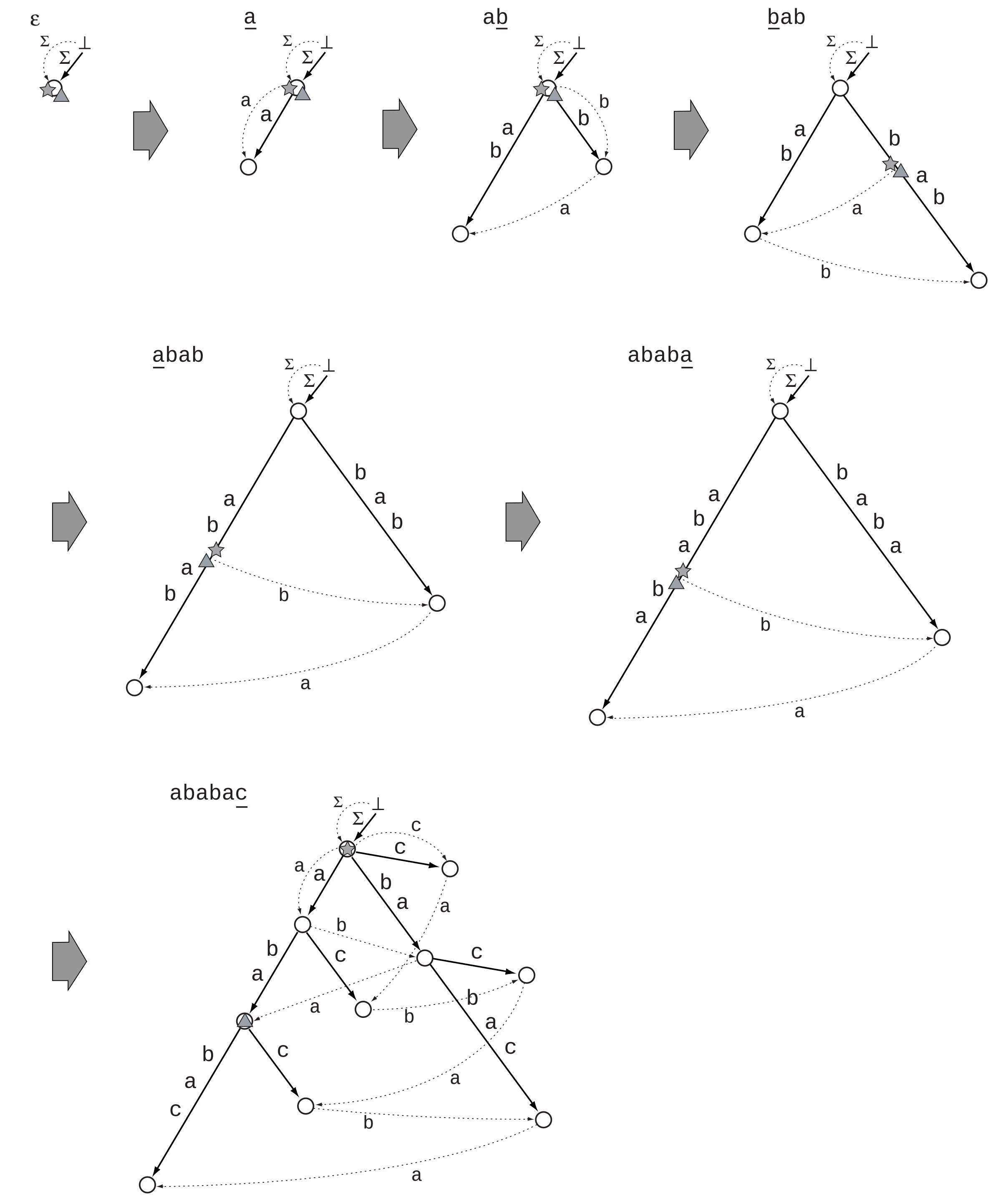


Figure 8.3: A bidirectional construction of $STree'(w)$ with $w =$ `ababac`. Solid arrows represent edges, and dotted arrows denote labeled reversed suffix links. On Right Extension, the labeled reversed suffix links are used for another direction, that is, as "normal" suffix links. In each suffix tree, the triangle (resp. star) indicates the location of the longest repeated prefix (resp. suffix). The character newly added in each step is underlined.

Chapter 9

Generic Construction of Index Structures

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 do so, much attention and effort has been paid so far [77, 55, 12, 73, 9, 10, 18]. From viewpoints of practice and algorithmics, it is also very important to construct an index structure in *on-line* manner, where, for example, we can obtain $STree'(wa)$ only with small change of appending new character a to $STree'(w)$. If it is off-line, we have to construct $STree'(wa)$ from scratch, even if we had $STree'(w)$ beforehand. 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* efficiently. 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 and space. Hence an algorithm that can *directly* build an index structure for a set of strings is truly helpful.

Each index structure is suitable to solve particular problems. For example, $STree(w)$ is optimal to find all occurrences of a given pattern p in a text w [26], DAWGs are a good structure to find the longest common substring of two strings [15], CDAWGs are ideal when we want to save memory space, since its space complexity is strictly smaller than those of the other index structures [10]. Therefore, we should need every index structure in order that we can deal with various sorts of problems. The matter is, however, that we then have to implement at least four different algorithms, as there exist four index structures.

We are thereby been motivated to get rid of this trouble, and finally succeed *unifying* 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 them. The algorithm is endowed with all the desired properties: it runs on-line and in linear time, and can apply to a set of strings. However, as an exception construction of suffix tries cannot always be done in linear time since they can require quadratic space.

A complete pseudo-code of our algorithm is shown in Figure 9.1, Figure 9.2, Figure 9.3 and Figure 9.4. We have marked each line with four symbols: there, $\clubsuit$ ($\spadesuit$, $\heartsuit$ and $\diamondsuit$, resp.) indicates the lines that can be executed when suffix tries (suffix trees, DAWGs, and CDAWGs, reps.) are constructed. It has succeeded revealing the essential common points and separating the small differences among the typical algorithms [73, 9, 39]. In fact, all the control blocks are exactly the same and all differences can be packed into the only *one* procedure in Figure 9.2 that creates a new edge. This means that we can choose which index structure to build, only by 'switching' the procedure.

Furthermore, by comparing the definitions of index structures given in Chapter 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.

This result was also reported in [38].

Our algorithm to be shown later on constructs $STree'(S)$ and $CDAWG'(S)$ rather than $STree(S)$ and $CDAWG(S)$, 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, and then we separately give an exposition for the different part.

9.1 Construction of an Index Structure for a Single String

We firstly consider the case that the input of the algorithm is a single string $w \in \Sigma^*$. Let $Index(w[1:i])$ be an arbitrary index structure of string $w[1:i]$ for $1 \le i \le |w|$. Our algorithm updates $Index(w[1:i])$ to $Index(w[1:i+1])$ by inserting the suffixes of $w[1:i+1]$ into $Index(w[1:i])$. Recall Definition 19 of the LRS, given in Section 4.1. Remember the suffixes of $w[1:i+1]$ can be divided into the following two groups, by $LRS(w[1:i+1])$.

```
♣♠♡♢   Algorithm Construction of an index structure on $\Sigma = \{w[-1], \dots, w[-m]\}$.
♣♠♡♢    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 9.1: Main routine of our algorithm. ♣ (♠, ♡ and ♢, resp.) indicates the lines that can be executed when suffix tries (suffix trees, DAWGs, and CDAWGs, reps.) are constructed.

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

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

The group (2) is empty in such case that $LRS(w[1 : i+1]) = \varepsilon$, 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 $Index(w[1 : i])$. Meanwhile, we insert each suffix of case (1) into $Index(w[1 : i])$, from $w[1 : i+1]$ to $w[j : i+1]$. Let us call *the start point* of $Index(w[1 : i+1])$ the location where $LRS(w[1 : i])$ is represented in $Index(w[1 : i+1])$, and call *the end point* of $Index(w[1 : i+1])$ the location where $LRS(w[1 : i+1])$ is represented in $Index(w[1 : i+1])$. 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 $1 \leq h' \leq j'$ where $LRS(w[j'+1 : i]) = w[1 : i]$.

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

The main routine of our algorithm is shown in Figure 9.1. In the algorithm, an edge (u, α, v) is represented by $(u, (k, p), v)$ such that k (resp. p) represents a beginning position (resp. an ending position) of the label in the input string w. The main routine calls **function** *update*, shown in Figure 9.2, each time a new character is scanned. **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 $LRS(w[1:i+1])$. This is examined by **function** *check_end_point*, in the 2nd line of **function** *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 Figure 9.2. The detail of the dependence will be mentioned in the sequel. The location from which the algorithm should insert each suffix of 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 $j'+1 \le h \le j-1$, which is in case (1-a). Then, in the 15th line of **function** *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 $LRS(w[1 : i+1])$, i.e. the end point, is found.

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

9.1.1 $\boxed{\textbf{CreateNewEdge}}$ in Case of Constructing 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 8 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 of $STrie(w[1 : i])$ 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.

9.1.2 $\boxed{\textbf{CreateNewEdge}}$ in Case of Constructing Suffix Trees

Assume that we now have $STree'(w[1 : i])$. Recall Definition 9. 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 the label of an edge directing a leaf node of $\mathit{STree}'(w[1:i])$ to "∞", as in Ukkonen's algorithm [73], and 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 Figure 1.1. 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 $\mathit{STree}'(w[1:i])$ 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 $\mathit{LRS}(w[h:i+1])$, 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 **function** *split_edge* called in the 10th line of **function** *update*.

9.1.3 $\boxed{\textbf{CreateNewEdge}}$ in Case of Constructing DAWGs

Assume that we now have $\mathit{DAWG}(w[1:i])$. Definition 10 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 $w[i+1:i+1]$ with 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 **function** *update* the active point was moved to the advanced point of $\mathit{DAWG}(w[1:i])$ 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 suffix tries to DAWGs. Suppose that the end point has already found, that is, the insertion of all 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 **function** *update*, **function** *separate_node* is called, which examines whether or not $w[j+1:i+1] = u$, where $u = \overset{w[1:i+1]}{\overleftarrow{w[j+1:i+1]}}$. 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 $[w[j+1:i+1]]^R_{w[1:i+1]}$.

9.1.4 $\boxed{\textbf{CreateNewEdge}}$ in Case of Constructing CDAWGs

Assume that we now have $CDAWG'(w[1:i])$. The way to update $CDAWG'(w[1:i])$ is like a 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 13 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" of DAWGs to CDAWGs. The new edge is directed to the sink node. It corresponds to the "minimization" of suffix trees to CDAWGs. 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 **function** *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. **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.

9.2 Extension to a Set of Strings

Given a set $S = \{w_1, w_2, \ldots, w_k\}$, we regard it as a sequence $t = w_1\$_1w_2\$_2\cdots w_k\$_k$, 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 Figure 9.1, 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 19 *For any set S of strings, the proposed algorithm constructs $STrie(S)$,*

$STree'(S)$, $DAWG(S)$, and $CDAWG'(S)$ on-line, with changing the 12th line of **function** *update accordingly. $STree'(w)$, $DAWG(w)$, and $CDAWG(w)$ are all constructed in $O(\|S\|)$ time.*

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 Figure 9.5, Figure 9.6, Figure 9.7, and Figure 9.8.

```
♣♠♡♢    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 ≤ 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     CreateNewEdge     /* Change only this line */
♣♠♡♢    13     if oldr ≠ nil then suf(oldr) := r;
♣♠♡♢    14     oldr := r;
♣♠♡♢    15     (s,k) := canonize(suf(s),(k,p−1));
♣♠♡♢    16 if oldr ≠ 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));
```

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) := ∞;
    create a new edge (r,(p,∞),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) := ∞;
    create a new edge (r,(p,∞),s_h);
```

Figure 9.2: Function *update*. ♣ (♠, ♡ and ♢, resp.) indicates the lines that can be executed when suffix tries (suffix trees, DAWGs, and CDAWGs, reps.) are constructed.

```
♣♠♡♢   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 9.3: Functions *check_end_point*, *extension*, and *canonize*. ♣ (♠, ♡ and ♢, resp.) indicates the lines that can be executed when suffix tries (suffix trees, DAWGs, and CDAWGs, reps.) are constructed.

```
 ♠ ♢   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' ≤ 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') ≠ canonize(s, (k,p));
 ♠ ♢    11 return (r', p + 1);
```

Figure 9.4: Other functions. ♣ (♠, ♡ and ♢, resp.) indicates the lines that can be executed when suffix tries (suffix trees, DAWGs, and CDAWGs, reps.) are constructed.

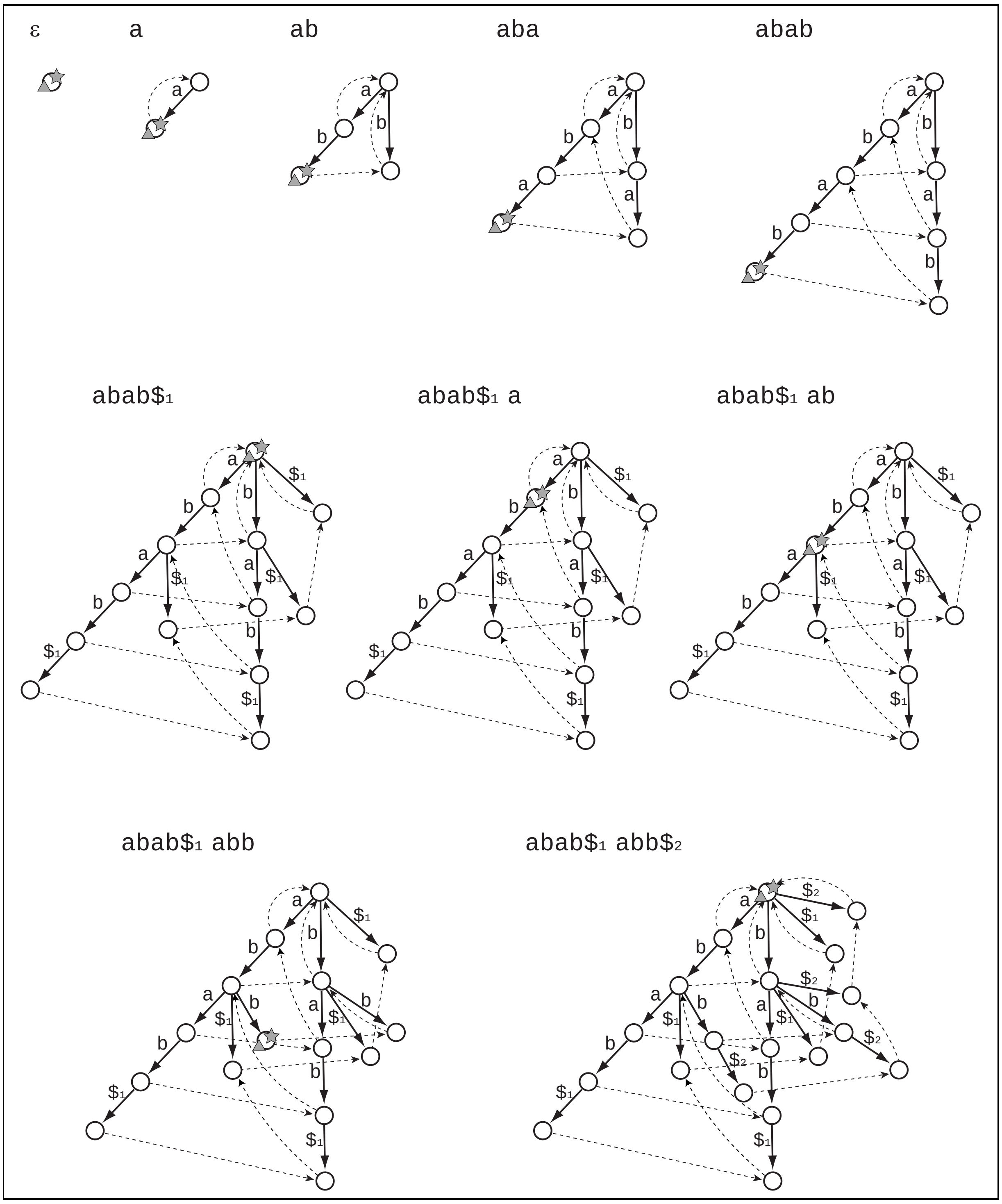


Figure 9.5: A snapshot of the on-line construction of $STrie(S)$ where $S = \{\texttt{abab}, \texttt{abb}\}$. The gray star and the triangle represent the active point and the advanced point of each step, respectively. The broken lines are suffix links.

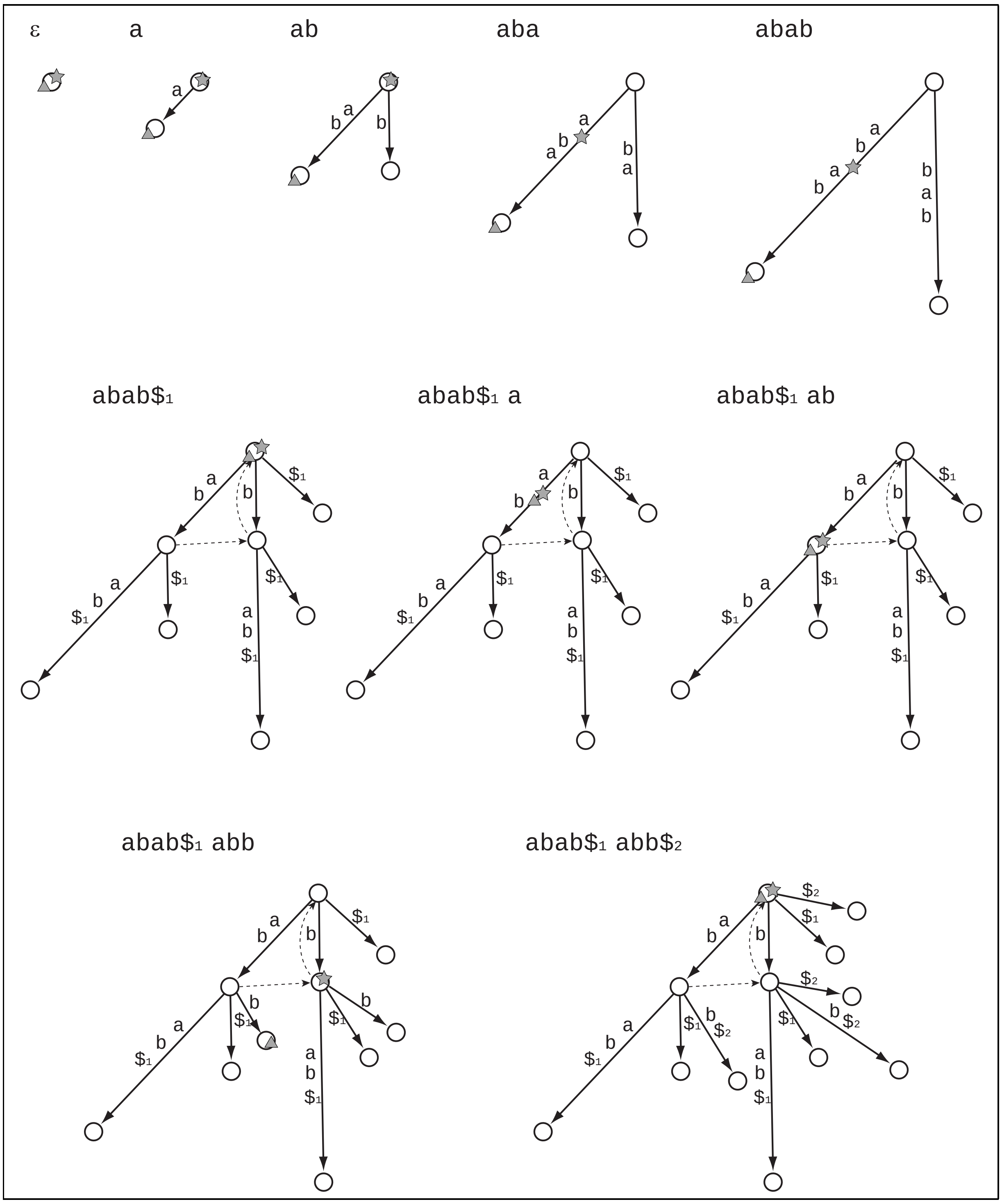


Figure 9.6: A snapshot of the on-line construction of $STree'(S)$ where $S = \{\texttt{abab}, \texttt{abb}\}$. The gray star and the triangle represent the active point and the advanced point of each step, respectively. The broken line are suffix links.

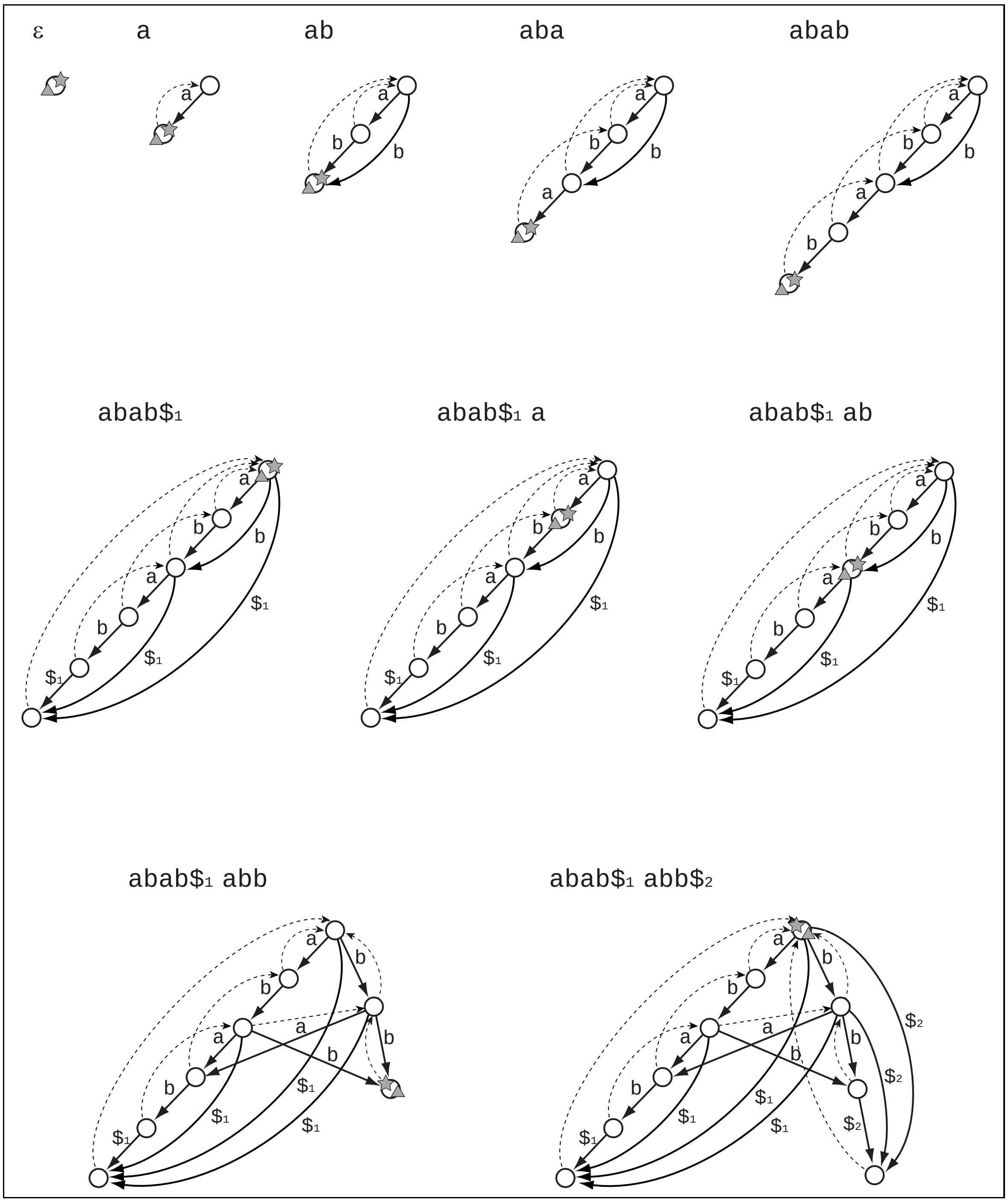


Figure 9.7: A snapshot of the on-line construction of *DAWG*(S) where $S = \{\texttt{abab}, \texttt{abb}\}$. The gray star and the triangle represent the active point and the advanced point of each step, respectively. The broken lines are suffix links.

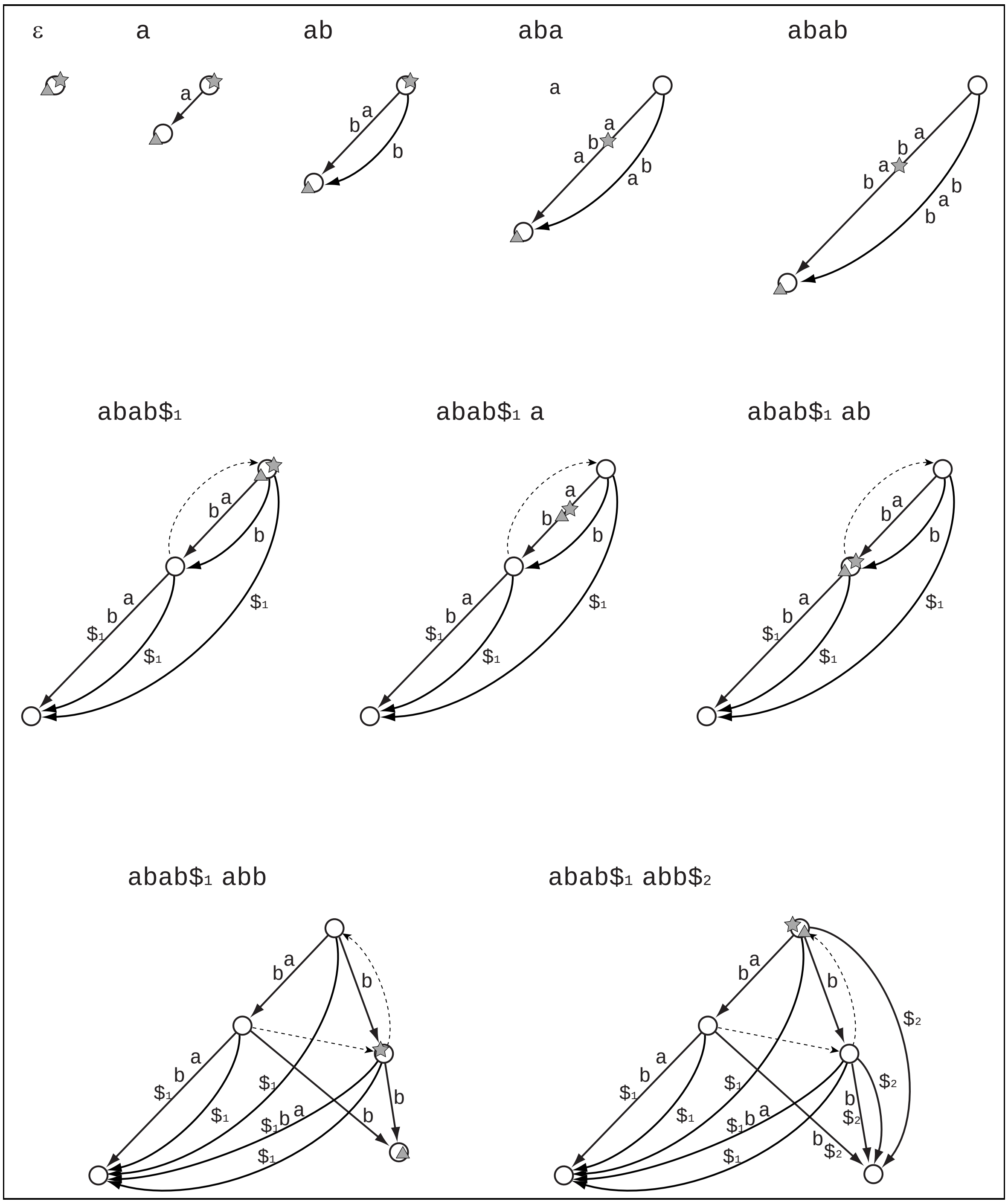


Figure 9.8: A snapshot of the on-line construction of *CDAWG*′(S) where $S = \{\texttt{abab}, \texttt{abb}\}$. The gray star and the triangle represent the active point and the advanced point of each step, respectively. The broken lines are suffix links.

Chapter 10

Other Pattern Matching Problems

We have discussed substring pattern matching problems in the previous chapters. This chapter introduces more advanced and difficult pattern matching problems.

10.1 Subsequence Pattern Matching

String $p \in \Sigma^*$ is said to be a *subsequence* of string $w \in \Sigma^*$ if p can be obtained by removing zero or more characters from w. For instance, let $w = \texttt{abba}$. Then $p = \texttt{aba}$ is a subsequence of w.

Definition 27 (Subsequence Pattern Matching Problem)
Instance: text $w \in \Sigma^*$ and pattern $p \in \Sigma^*$.
Determine: whether p is a subsequence of w.

Obviously, this is a generalization of the substring pattern matching problem of Definition 7. It is almost trivial to show that the above problem is $O(|w| + |p|)$-time solvable by the use of a DFA which accepts all strings which contain p as a subsequence [69]. On the other hand, an index structure for the subsequence pattern matching problem was proposed by Baeza-Yates [5], which is called *directed acyclic subsequence graphs* (*DASGs*). The DASG of w, denoted by $DASG(w)$, is the smallest DFA that accepts all subsequences of w, and thus, it enables us to solve the above problem in $O(|p|)$ time. $DASG(w)$ has $|w| + 1$ nodes, and all of them are accepting nodes.

$DASG(w)$ with $w = \texttt{abba}$ is shown in Figure 10.1.

Theorem 20 (Baeza-Yates [5]) *For any $w \in \Sigma^*$, $DASG(w)$ can be constructed in $O(|\Sigma| \cdot |w|)$ time.*

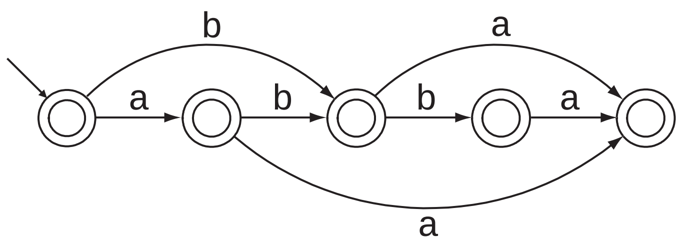


Figure 10.1: $DASG(w)$, where $w = \texttt{abba}$.

Baeza-Yates presented a linear-time algorithm for construction of DASGs, which processes a given string from right to left. On the other hand, Troníček and Melichar [71] introduced a left-to-right algorithm for building DASGs in linear time. Construction of DASGs for sets of strings was also considered in [5, 31, 17]

10.2 Episode Pattern Matching

An *episode pattern* consists of a pair $\langle p, k\rangle$, where $p \in \Sigma^*$ and $k \in \mathcal{N}$. Episode pattern $\langle p, k\rangle$ is said to *match* $w \in \Sigma^*$ if p is a subsequence of some substring u of w and $|u| \leq k$ [54]. For instance, let $w = \texttt{abba}$. Then $\langle \texttt{aa}, 4\rangle$ matches w, but $\langle \texttt{aa}, 3\rangle$ not.

Definition 28 (Episode Pattern Matching Problem)
Instance: text $w \in \Sigma^*$ and episode pattern $\langle p, k\rangle \in \Sigma^* \times \mathcal{N}$.
Determine: whether $\langle p, k\rangle$ matches w.

Note that the episode pattern matching problem with $\langle p, |p|\rangle$ is the same as the substring pattern matching problem with p. Also, the episode pattern matching problem with $\langle p, \infty\rangle$ is the same as the subsequence pattern matching problem with p.

Mannila et al. [54] gave an algorithm that solves the above problem in $O(|w| \cdot |p|)$ time. An index structure for efficiently solving the episode pattern matching problem was proposed by Troníček [70], which is called *episode directed acyclic subsequence graphs* (*EDASGs*). The EDASG of w, denoted by $EDASG(w)$ has two kinds of edges, forward edges and backward edges. The forward edges correspond to the edges of $DASG(w)$, the backward edges $DASG(w^{\mathrm{rev}})$. Each node of $EDASG(w)$ is designated by an integer $0 \leq i \leq |w|$ representing the node number.

Theorem 21 (Troníček [70]) *For any $w \in \Sigma^*$, $EDASG(w)$ can be constructed in $O(|\Sigma| \cdot |w|)$ time.*

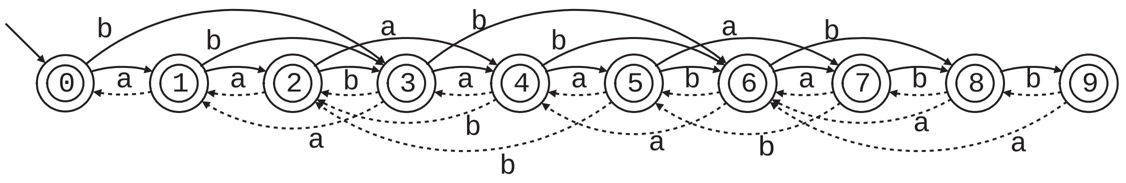


Figure 10.2: *EDASG*(w), where $w =$ `aabaababb`. Solid arrows denote the forward edges, and broken arrows denote the backward edges.

EDASG(w) with $w =$ `aabaababb` is shown in Figure 10.2. In order to examine if episode pattern $\langle$`abb`$, 4\rangle$ matches $w =$ `aabaababb` or not, we begin with the initial node 0 and then arrive at node 6, by traversing the forward edges spelling out `abb`. It means that the shortest substring of w beginning at position 1 and containing `abb` as a subsequences is $w[1 : 6] =$ `aabaab`. Moreover, the number obtained by summing the node numbers 6 and 0 corresponds to the length of the matched substring `aabaab`, that is, $6 - 0 = |$`aabaab`$|$. Notice that the obtained number 6 does not always represent the length of the shortest substring of w that ends at position 6 and contains `abb` as a subsequence. In fact, string `abaab`, which is a suffix of `aabaab`, is the one. To find the string `abbab`, we move from node 6 with the backward edges spelling out `bba`, the reversal of the given pattern string, and then reach node 1. As a result, the suffix of `aabaab` of length $6-1 = 5$ is the shortest substring containing `abb` in the range from position 1 to position 6 in w. Still, since $5 > 4$, we have to examine other ranges. To do so, we continue the same traversal starting from node 2, that is the next node of node 1. By the forward traversal spelling out `abb`, we reach node 8, and then the backward traversal spelling out `bba` takes us to node 4. This time, the found substring `abab` contains the subsequence `abb`, and the length $8 - 4 = 4$ does not exceed the threshold value given. This way, we get to know the episode pattern $\langle$`abb`$, 4\rangle$ matches `aabaababb`.

It is quite obvious that the examination of whether a given episode pattern $\langle p, k\rangle$ matches w with *EDASG*(w) still takes $O(|w| \cdot |p|)$ time, but it seems to be practically faster than a naive method with the use of the standard edit distance table.

10.3 VLDC Pattern Matching

Let $\star$ be a *wildcard* that matches *any* string in Σ^*. The wildcard $\star$ is also called the *variable-length-don't-care* symbol. Let $\Pi = (\Sigma \cup \{\star\})^*$. An element $q \in \Pi$ is called a *variable-length-don't-care pattern* (*VLDC pattern*). It is also called a *regular pattern* in the context of machine learning, such as in [65]. A VLDC pattern q is said to *match* string $w \in \Sigma^*$ if w is obtained by replacing $\star$'s in q with some strings in Σ^*. Assume $\mathtt{a}, \mathtt{b} \in \Sigma$. Then $\mathtt{ab} \star \mathtt{bb} \star \mathtt{ba}$ is an example of a VLDC pattern and, for instance, matches string `abbbbaaaba` with the first and second $\star$'s replaced by `b` and `aaa`, respectively.

Definition 29 (VLDC Pattern Matching Problem)
Instance: text $w \in \Sigma^*$ and VLDC pattern $q \in \Pi$.
Determine: whether q matches w.

Note that this is a generalization of the substring pattern matching problem of Definition 7. For instance, consider a pattern string $\mathtt{abc} \in \Sigma^*$. The substring matching problem with `abc` exactly corresponds to the VLDC pattern matching with $\star\mathtt{abc}\star$. Also, VLDC pattern $\star\mathtt{a} \star \mathtt{b} \star \mathtt{c}\star$ leads to the subsequence pattern matching problem with `abc`.

Our first idea for efficient solution of the VLDC pattern matching is to use a DFA for a given VLDC pattern $q \in \Pi$. We construct a DFA that accepts all strings q matches, and run it over a given text string $w \in \Sigma^*$. It is not difficult to see that such a DFA can be constructed in $O(|q|)$ time, and the running time is $O(|w|)$.

To solve the above problem efficiently in case that w is fixed and q is flexible, we introduce an index structure called *wildcard directed acyclic word graphs* (*WDAWGs*). The WDAWG of w, denoted by $\mathit{WDAWG}(w)$, is the smallest automaton that recognizes all VLDC patterns which match w. Therefore, using $\mathit{WDAWG}(w)$ enables us to solve the VLDC pattern matching problem in $O(|q|)$ time. As for the size and construction of WDAWGs, we have the following theorems.

Theorem 22 *When $|\Sigma| \geq 2$, the number of nodes of $\mathit{WDAWG}(w)$ for a string w is $\Theta(|w|^2)$. It is $\Theta(|w|)$ for a unary alphabet.*

Theorem 23 *For any string $w \in \Sigma^*$, $\mathit{WDAWG}(w)$ can be constructed in time linear in its size.*

$\mathit{WDAWG}(w)$ with $w = \mathtt{abbab}$ is shown in Figure 10.3.

We remark that WDAWGs are inherently the same as *minimum all-suffixes directed acyclic word graphs* (*MASDAWG*) that will be introduced in Chapter 11.

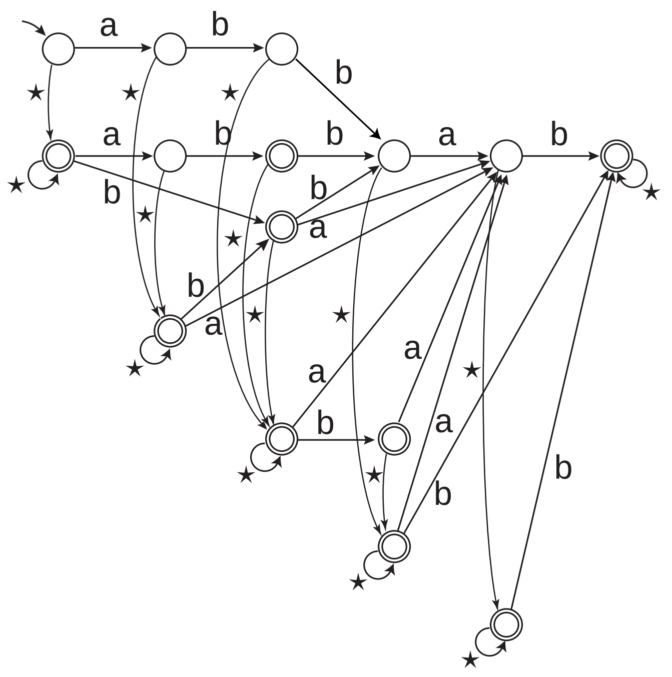


Figure 10.3: *WDAWG*(w) with $w =$ `abbab`.

10.4 VLDC Pattern Matching within a Window

We here consider a natural extension of the VLDC pattern matching problem with the introduction of an integer k called the *window size*. Let $q \in \Pi$ and $q[i], q[j]$ be the first and last characters in q that are not $\star$, respectively, where $1 \leq i \leq j \leq |q|$. If q matches $w \in \Sigma^*$, let $w[i'], w[j']$ be the characters to which $q[i]$ and $q[j]$ correspond, respectively, where $1 \leq i' \leq j' \leq |w|$. (Note that we might have more than one combination of i' and j'.) If there exists a pair of i' and j' that satisfies $j' - i' + 1 \leq k$, we say that q *occurs* in w within a window of size k. Then the pair $\langle q, k\rangle$ is said to *match* w. For instance, pair $\langle \star\mathtt{ab}\star\mathtt{ab}\star, 6\rangle$ matches string `babbbabb`, but pair $\langle \star\mathtt{ab}\star\mathtt{ab}\star, 5\rangle$ does not match the string.

Definition 30 (VLDC Pattern Matching Problem in Window)
Instance: text $w \in \Sigma^*$ and VLDC pattern with window size $\langle q, k\rangle \in \Pi \times \mathcal{N}$.
Determine: whether $\langle q, k\rangle$ matches w.

We here show our three approaches to efficiently solve the above problem. The first is to adopt the standard dynamic programming method. For a string $w \in \Sigma^*$ and a pattern $q \in \Pi$ with, let d_{ij} be the length of the shortest suffix of $w[1 : j]$ that $q[1 : i]$ matches, where $0 \leq i \leq |q|$ and $0 \leq j \leq |w|$. Computation of all d_{ij}'s can be done in $O(|w| \cdot |q|)$

time, based on the following recurrences: $d_{00} = 0$,

$$d_{0j} = \begin{cases} 0 & \text{if } q[1] = \star \\ \infty & \text{otherwise} \end{cases} \quad \text{for } j \geq 1,$$

$$d_{i0} = \begin{cases} d_{i-1,0} & \text{if } q[1] = \star \\ \infty & \text{otherwise} \end{cases} \quad \text{for } i \geq 1, \text{ and}$$

$$d_{ij} = \begin{cases} \min\{d_{i-1,j-1}+1, d_{i,j-1}+1, d_{i-1,j}\} & \text{if } q[i] = \star \\ d_{i-1,j-1}+1 & \text{if } q[i] = w[j] \\ \infty & \text{otherwise} \end{cases} \quad \text{for } i \geq 1 \text{ and } j \geq 1.$$

Then $\theta_{w,q} = \begin{cases} \min_{1 \leq j \leq n}\{d_{mj}\} & \text{if } q[m] = \star \\ d_{mn} & \text{otherwise.} \end{cases}$

The second approach is to preprocess a given VLDC pattern $q \in \Pi$. We construct a DFA accepting all strings q matches, and another DFA accepting all strings q^{rev} matches. We run these DFA over a given string $w \in \Sigma^*$. If $q[1] \neq \star$ ($q[m] \neq \star$, respectively), we have only to compute the shortest prefix (suffix, respectively) of w that q matches and return its length ℓ. If $\ell \leq k$, the pair $\langle q, k \rangle$ matches w, and otherwise, not. We now consider the case $q[1] = q[m] = \star$. Firstly, we run the DFA for q over w. Suppose that q is recognized between positions i and j in w, where $1 \leq i < j \leq |w|$ and $j - i > |q|$. A delicate point is that it is unsure whether $w[i:j]$ corresponds to the shortest occurrence of q ending at position j. How can we find the shortest one? It can be found by running the DFA for q^{rev} *backward*, over w from position j. Assume that q^{rev} is recognized at position h, where $i \leq h < j - |q|$. Then $w[h:j]$ corresponds to the shortest occurrence of q ending at position j. If $j - h + 1 \leq k$, we stop here and know that the pair $\langle q, k \rangle$ matches w. Otherwise, we resume the running of the DFA for q from position $h+1$, and continue the above procedure until either finding an occurrence of q short enough, or encountering the last node $|w|$ without finding any appropriate occurrence. This approach is feasible in $O(|w| \cdot |q|)$ time.

The third approach is to preprocess a text string $w \in \Sigma^*$, i.e., we construct $\mathit{WDAWG}(w)$ and $\mathit{WDAWG}(w^{\text{rev}})$. For any $w \in \Sigma^*$, each and every node of $\mathit{WDAWG}(w)$ can be designated by a position in w. Thus we can perform a procedure similar to the second approach above. It also takes $O(|w| \cdot |q|)$ time in total.

Chapter 11

Minimum All-Suffixes DAWGs

This chapter presents a new kind of index structure called *minimum all-suffixes directed acyclic word graphs* (*MASDAWGs*). We begin this chapter with introducing two new string matching problems that are efficiently solvable by the use of MASDAWGs.

We define a *beginning-sensitive pattern* (*BS-pattern*) to consist of a pair $\langle p, i\rangle$ such that $p \in \Sigma^*$ and $i \in \mathcal{N}$.

Definition 31 (BS-Pattern Matching Problem)
Instance: text $w \in \Sigma^*$ and BS-pattern $\langle p, i\rangle \in \Sigma^* \times \mathcal{N}$.
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 $O(|p|)$ time for an arbitrary pair $\langle p, i\rangle$ by using the DAWGs for all suffixes of w. If $i > |w|$, the BS-pattern never matches w. Otherwise, we begin with the source node of $DAWG(w[i:])$ and start the matching of p. If p is recognized, the BS-pattern is said to *match* w, and otherwise, not. It is clear that this way the above problem is solved in $O(|p|)$ time.

A *region-sensitive pattern* (*RS-pattern*) consists of a triple $\langle p, (i, j)\rangle$ where $p \in \Sigma^*$ and $i, j \in \mathcal{N}$.

Definition 32 (RS-Pattern Matching Problem)
Instance: text $w \in \Sigma^*$ and RS-pattern $\langle p, (i, j)\rangle \in \Sigma^* \times \mathcal{N} \times \mathcal{N}$.
Determine: whether p is a substring of $w[i:j]$.

This is a natural extension of the BS-pattern matching problem in which $j = |w|$. To solve this problem, we again use the DAWGs of all suffixes of w. For every $0 \leq h \leq |w|$, each node $[x]^R_w$ of $DAWG(w[h:])$ is assigned with the position of the *right most occurrence* of x in $w[h:]$. For RS-pattern $\langle p,(i,j)\rangle$, if $i > |w|$, the RS-pattern never matches w. Otherwise, we start with the source node of $DAWG(w[i:])$ and examine whether or not p is recognized. If it is recognized, we compare j with the number k stored in the node at which p finally arrived. If $j \leq k$, the RS-pattern *matches* w, and otherwise, not. Obviously, the problem can be solved in $O(|p|)$ time.

Now we are motivated to solve these two pattern matching problems using only *one* structure, with as little computational space as possible. We will indeed introduce such data structures in the following sections, called *minimum all-suffixes directed acyclic word graphs* (*MASDAWGs*).

The result was originally presented in [43].

11.1 All-Suffixes Directed Acyclic Word Graphs

Definition 33 $ASDAWG(w)$ is a kind of deterministic automaton with $|w|+1$ initial nodes, designated by integers $0,1,\ldots,|w|$, in which the subgraph consisting of the nodes reachable from an initial node 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 $ASDAWG(w)$, which is what we call the *naive* $ASDAWG(w)$. The number of nodes of the naive $ASDAWG(w)$ is $\Theta(|w|^2)$, simply because the size of $DAWG(w)$ is $O(|w|)$ (see Theorem 3). What we obtain by minimizing the naive $ASDAWG(w)$ is called the *minimum* $ASDAWG(w)$, denoted by $MASDAWG(w)$. The naive $ASDAWG(\texttt{abba})$ and $MASDAWG(\texttt{abba})$ are shown in Figure 11.1. It is emphasized that $MASDAWG(w)$ is the minimum index structure to solve in $O(|p|)$ time the two pattern matching problems proposed above. Moreover, $WDAWG(w)$ introduced in Chapter 10 can be easily constructed from $MASDAWG(w)$, and thus, MASDAWGs contribute to an efficient solution to the VLDC pattern matching problem as well.

The minimization of the naive $ASDAWG(w)$ 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 $x \in \mathit{Substr}(u)$. The equivalence relation, denoted by $\sim_w$, is defined by

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

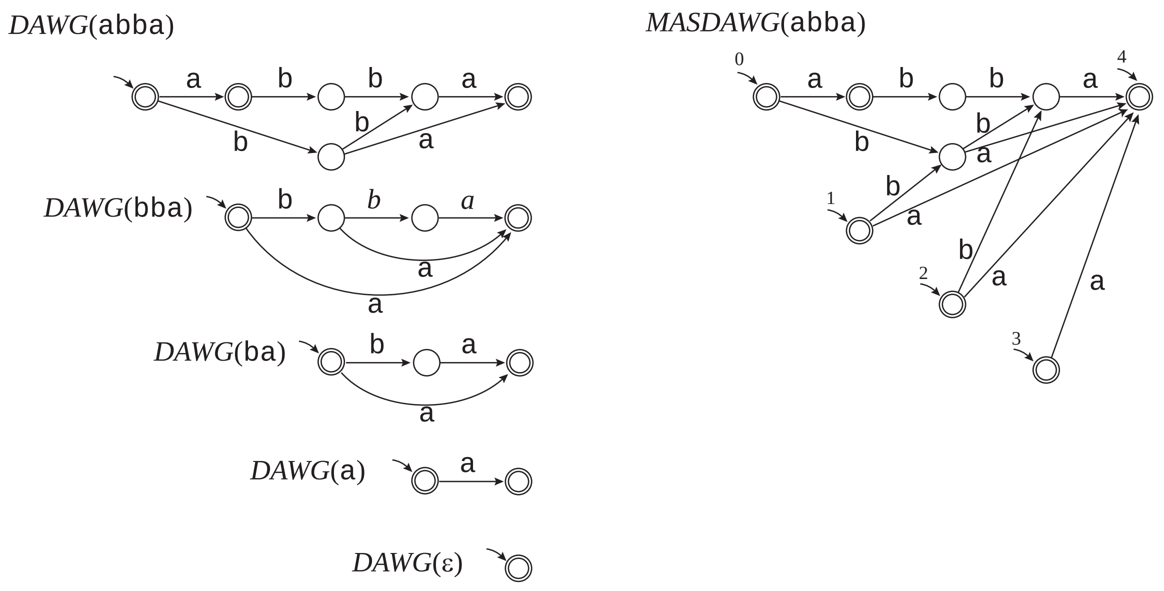


Figure 11.1: On the left, *DAWG*(x) for $x \in$ *Suffix*(abba) are shown, and the collection of those DAWGs is the naive *ASDAWG*(w). On the right *MASDAWG*(abba) is displayed. While there are 16 nodes and 16 edges in the former in total, there are 9 nodes and 12 edges in the latter. For example, the 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 27), and therefore the five are merged into one sink node.

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 16 *Let $u \in$ Suffix(w). Let x be a non-empty substring 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

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

In case more than one string exist in $[x]^R_u$, the chain length r is maximized by choosing 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 26 *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 $x \in Substr(u)$ with $x \neq \varepsilon$, and $y \in Substr(tau)$ with $y \neq \varepsilon$. Assume $x^{-1}Suffix(u) = y^{-1}Suffix(tau)$. We have two cases to consider.

- $x \equiv^R_u y$. In this case, every occurrence of the string y within tau must be included within the u part. Thus, we have $x^{-1}Suffix(u) = y^{-1}Suffix(au)$.
- $x \not\equiv^R_u y$. 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 $x^{-1}Suffix(u) = z^{-1}Suffix(au)$.

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 node.

Lemma 27 *Suppose $b \in \Sigma$ and $u, bu \in Suffix(w)$. Let $y \in Substr(bu)$ and assume y is the representative of $[y]^R_{bu}$. 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 Figure 11.2.

As an extreme case of Lemma 27 where $\ell = 0$, the node $[\varepsilon]^R_\varepsilon$ of $DAWG(\varepsilon)$ is always equivalent to the sink node $[b]^R_b$ of the previous $DAWG(b)$.

Next, we consider the case of the nodes other than the initial node.

Lemma 28 *Suppose $b \in \Sigma$ and $u, bu \in Suffix(w)$. Let $x \in Substr(u)$ with $x \neq \varepsilon$. Let $y \in Substr(bu)$ with $y \neq \varepsilon$. Assume x and y are the representatives of $[x]^R_u$ and $[y]^R_{bu}$, respectively. The equivalence $\langle u, [x] \rangle \sim_w \langle bu, [y] \rangle$ implies that if $y \in Prefix(bu)$ then*

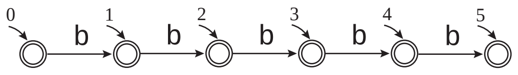


Figure 11.2: *MASDAWG*(w) for $w = \mathtt{b}^5$. For every $i = 0, 1, \ldots, 4$, the initial node $[\varepsilon]^R_{\mathtt{b}^i}$ of *DAWG*$(\mathtt{b}^i)$ is equivalent to the node $[\mathtt{b}]^R_{\mathtt{b}^{i+1}}$ of *DAWG*$(\mathtt{b}^{i+1})$.

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

1. When $y \in$ *Prefix*(bu). Then, $i = |y| - 1$ and $y = by'$ with $y' = u[1:i]$. Since $u[i+1:] \in$ *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 $y' \notin$ *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 $y' \in$ *Prefix*(u), y' must be the representative of $[y']^R_u = [x]^R_u$, thus we have $x = y'$.

   (b) When $y' \in$ *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 $u[j+1:] \in x^{-1}$*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 $x \in$ *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 $x \equiv_u^R y$. 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 $x^{-1}\mathit{Suffix}(bu) \neq x^{-1}\mathit{Suffix}(u)$. 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 $bu = h\delta yu[j+1:]$, which implies $u[j+1:] \in y^{-1}\mathit{Suffix}(bu)$, a contradiction. Hence we have $x \equiv_{bu}^R y$. 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 28 fit to Proposition 16, whereas Case 3 is irregular in the sense that the two equivalence classes $[x]_u^R$ and $[y]_{bu}^R$ have no common member despite $\langle u, [x]\rangle \sim_w \langle bu, [y]\rangle$. See Figure 11.1, which includes instances of Case 1, Case 2, and Case 3.

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

Theorem 24 *When $|\Sigma| \geq 2$, the number of nodes of 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 27. 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 $x \not\equiv_{u_i}^R ax$ and $x \not\equiv_{u_i}^R b^{-1}x$, and therefore $[x]_{u_i}^R = \{x\}$. Thus x is the representative of $[x]_{u_i}^R$, 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 *DAWG*(u_i) is the owner of the node corresponding to $[x]_{u_i}^R$. Thus, the nodes of *MASDAWG*(w) corresponding to

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

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

11.2 On-Line Construction of MASDAWGs

Since the construction of the naive *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

$ASDAWG(w)$ (see [60]), we can build $MASDAWG(w)$ in $O(|w|^2)$ time. On the other hand, we have shown that the number of nodes in $MASDAWG(w)$ is $\Theta(|w|^2)$. We are therefore interested in on-line and direct construction of $MASDAWG(w)$. We have obtained the following result.

Theorem 25 *$MASDAWG(w)$ 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 11.3 illustrates the on-line construction of $MASDAWG(\texttt{abbab})$.

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

11.2.1 Suffix Links

In construction of MASDAWGs, 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 26, so that we can handle them at once. This is critical to 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).

11.2.2 Compact Representation of Node Length Information

Remember that, in on-line construction of the DAWG for a single string, there occurs an event so-called node separation, as mentioned in Section 4.3. Let $z = LRS(w)$. A node separation happens iff z is not the representative of $[z]^R_w$. The node $[z]^R_w$ can be detected

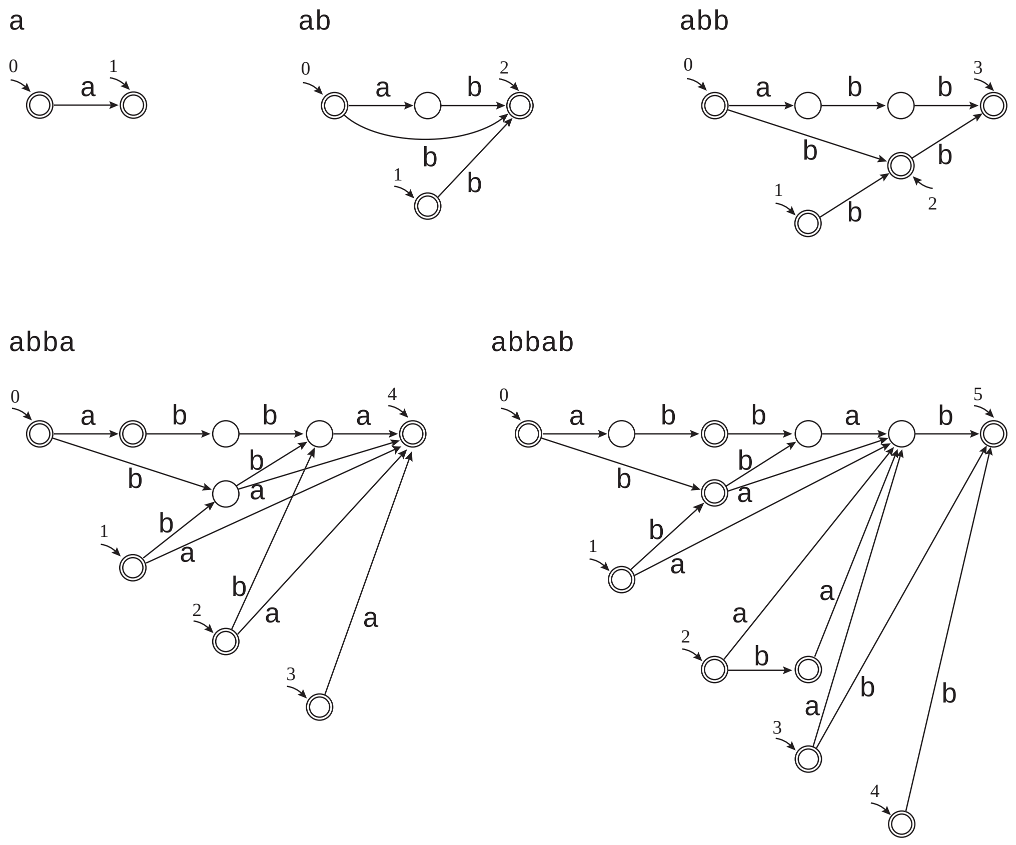


Figure 11.3: On-line construction of *MASDAWG*(w) for w = `abbab`. Each initial node becomes independent whenever the newly appended character violates the condition of Lemma 27. Node separation of other type occurs only twice. One happens during the update of *MASDAWG*(`ab`) to *MASDAWG*(`abb`). The 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 32.) (Note: Though each accepting node 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.)

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 out-going 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 thus 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 in 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 29 *Let $\langle w[i+1:], [x_1]\rangle, \ldots, \langle w[i+\ell:], [x_\ell]\rangle$ be 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 representative 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 Figure 11.4.)

Proof. By Lemma 28. □

For example, *MASDAWG*$(\texttt{abb})$ in Figure 11.3 has a node consisting of $\langle \texttt{abb}, [\texttt{b}]\rangle$ and $\langle \texttt{bb}, [\texttt{b}]\rangle$. Also, *MASDAWG*$(\texttt{abba})$ has a node consisting of $\langle \texttt{abba}, [\texttt{abb}]\rangle$, $\langle \texttt{bba}, [\texttt{bb}]\rangle$, and $\langle \texttt{ba}, [\texttt{b}]\rangle$.

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.

11.2.3 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 30 *Suppose $b \in \Sigma$ and $u, bu \in$ Suffix(w). Let $x \in$ Substr(u) with $x \ne \varepsilon$. Let $y \in$ Substr(bu) with $y \ne \varepsilon$. Assume x and y are the representatives of $[x]^R_u$ and $[y]^R_{bu}$,*

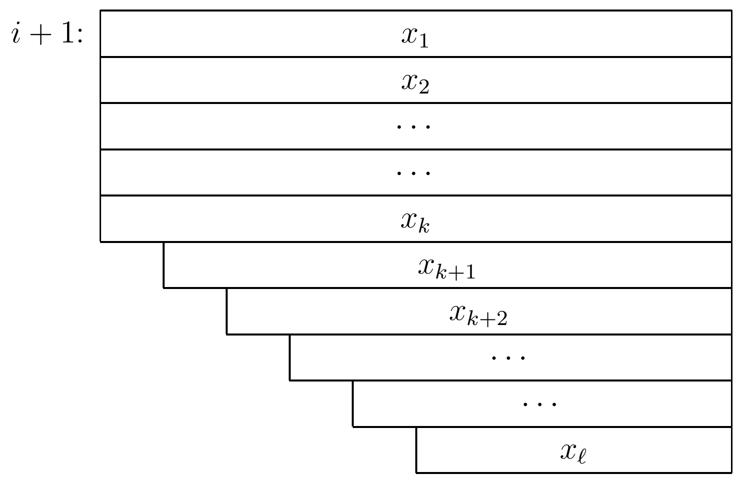


Figure 11.4: The representatives x_j of $[x_j]^R_{w[i+j:]}$ such that the nodes $\langle w[i+j:], [x_j]\rangle$ of the naive $ASDAWG(w)$ are merged into one node of $MASDAWG(w)$.

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 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 31 *Consider the node of $MASDAWG(w)$ stated in Lemma 29 (see Figure 11.4). 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 $w[i+j:], \ldots, w[i+\ell:]$.*

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 $w[i+j:], \ldots, w[i+t:]$. That is, for each $j = 1, \ldots, t$, 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 $\langle w[i+j:]a, [x_1]\rangle, \ldots, \langle w[i+\ell:]a, [x_\ell]\rangle$ are equivalent under $\sim_{wa}$, and the new nodes $\langle w[i+j:]a, [z]\rangle, \ldots, \langle w[i+t:], [z]\rangle$ 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 overcome is that there is another kind of node separation in the update of *MASDAWG*(w).

Lemma 32 *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 or 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 a concrete example of the first case of the above lemma, consider the update of *MASDAWG*(w) to *MASDAWG*(wa) with $w =$ `bbbbbab` and $a =$ `b`, which can be found in Figure 11.5 and Figure 11.6.

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.

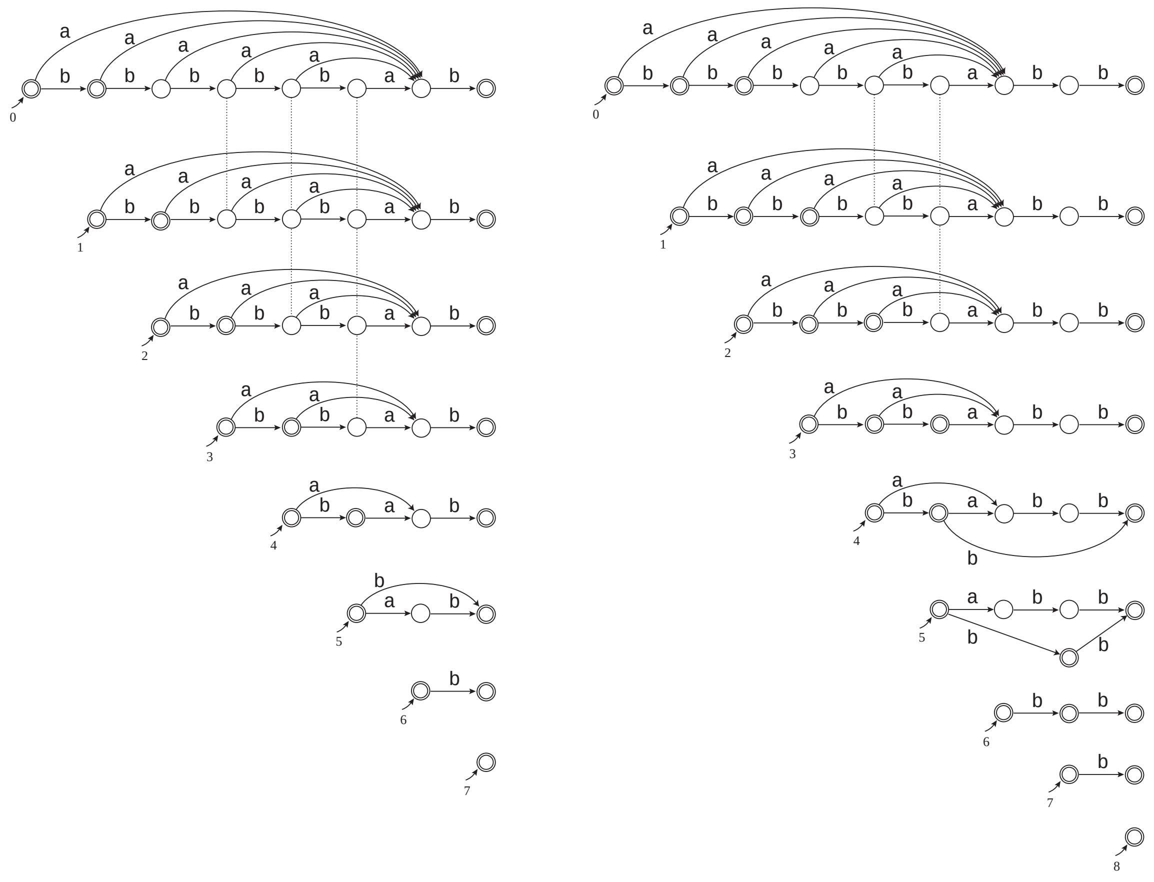
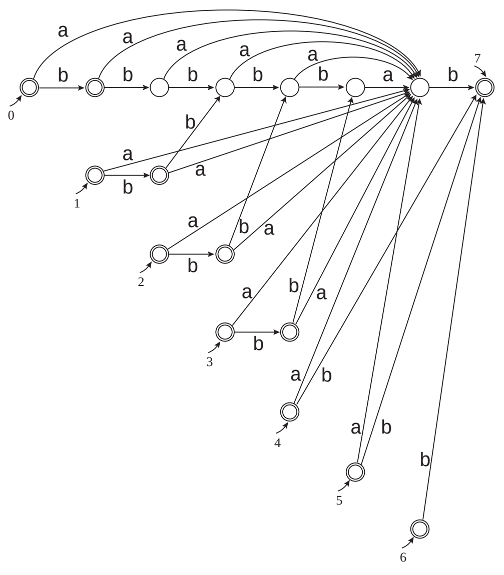


Figure 11.5: The naive *ASDAWG*(`bbbbbab`) on the left, and the naive *ASDAWG*(`bbbbbabb`) on the right. The nodes connected by the broken lines are equivalent due to Case 3. Recall the value of d mentioned in Lemma 32. In string `bbbbbab` the value of d is 1, whereas in string `bbbbbabb` $d = 2$ since the new `b` is added afterward.



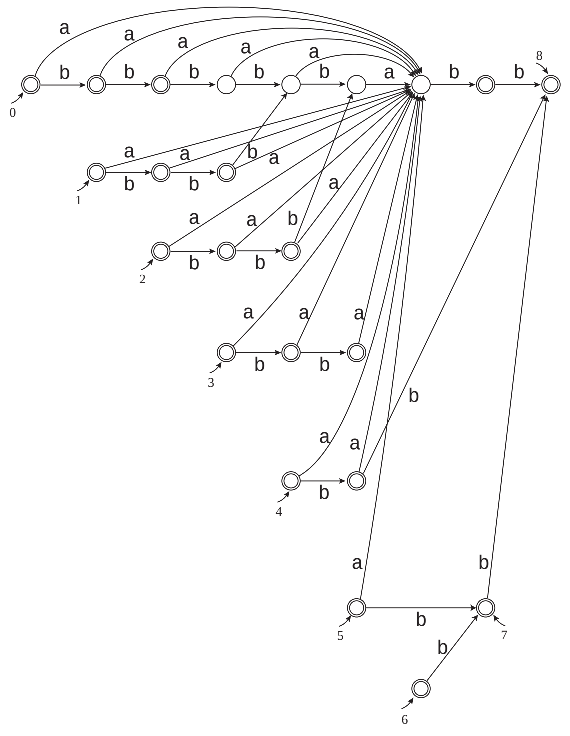


Figure 11.6: *MASDAWG*(`bbbbbab`) is on the left, and *MASDAWG*(`bbbbbabb`) is on the right. Compare the update of *MASDAWG*(`bbbbbab`) to *MASDAWG*(`bbbbbabb`) with that of the naive *ASDAWG*(`bbbbbab`) to the naive *ASDAWG*(`bbbbbabb`) shown in Figure 11.5.

Chapter 12

Space-Economical Construction of MASDAWGs

Chapter 11 was devoted to the introduction of MASDAWGs that are highly useful for several advanced pattern matching. On-line algorithm which, for any input string $w \in \Sigma^*$, directly constructs $MASDAWG(w)$ in time linear in the output size was also given in the previous chapter. The linearity and efficiency of the algorithm are highly dependent on the use of *suffix links*, kinds of failure transitions. Suffix links have been used for almost all time-efficient algorithms constructing index structures (e.g., see [77, 55, 73, 9, 10, 18, 16, 26]). On the other hand, it is also the fact that the memory space required by the implementation of suffix links is non-ignorable. Moreover, for each node of $MASDAWG(w)$, the algorithm of Chapter 11 additionally requires to keep the length of the longest string that reaches to the node, in the construction phase (see Section 11.2.2). Once completing the construction of $MASDAWG(w)$, these values are no longer necessary since examining whether or not a given pattern p occurs in the specified suffix of w can be done without them. In the sense of saving memory space needed for building MASDAWGs, therefore, the algorithm is not very efficient.

This chapter, on the other hand, presents a new algorithm to construct MASDAWGs *without suffix links nor length information*, which thus permits us to save considerable amount of memory space. This new algorithm is best understood as one constructing MASDAWGs by reading given strings from right to left. Namely, it builds $MASDAWG(aw)$ by adding necessary nodes and edges to $MASDAWG(w)$. Note that, in the contrast with this, the algorithm we presented in the previous chapter updates $MASDAWG(w)$ to $MASDAWG(wa)$.

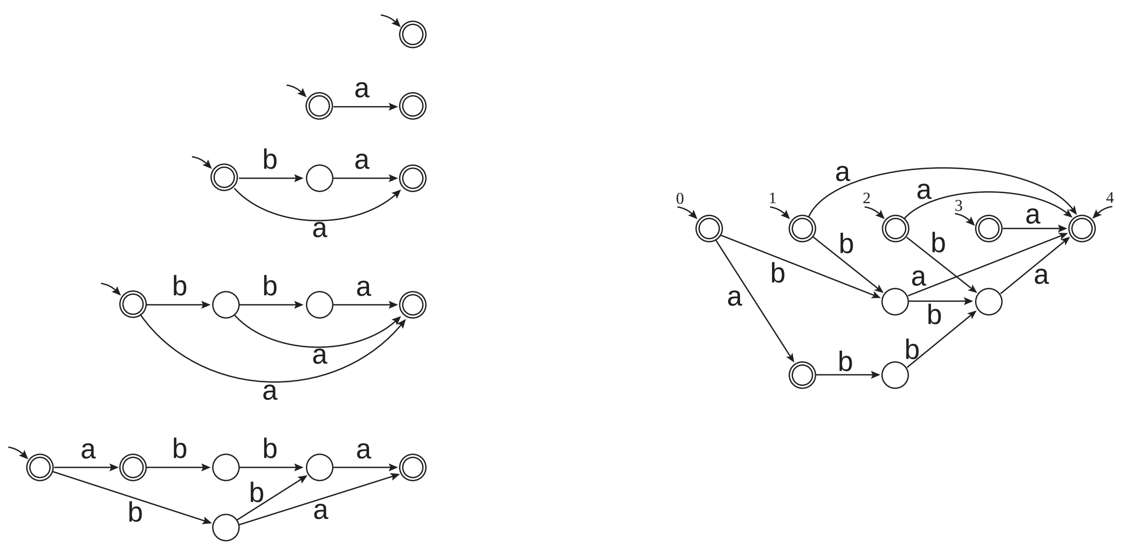


Figure 12.1: The naive *ASDAWG*(w) on the left, where w = `abba`. *MASDAWG*(w), on the right.

Furthermore, we reduce the space requirement by *compacting* the structure itself. We focus on *CDAWGs* whose space requirement is strictly smaller than that of DAWGs, both theoretically and practically [10, 18]. The all-suffixes version, named *minimum all-suffixes compact directed acyclic word graphs* (*MASCDAWGs*), is proposed in this chapter as well. We also present an algorithm that constructs *MASCDAWG*(w) in linear time with respect to its size, without using suffix links nor length information. This algorithm also processes w from right to left.

These results were primarily published in [42].

12.1 Space-Economical Construction of MASDAWGs

In this section we propose an algorithm for space-economical construction of MASDAWGs. The algorithm is designed to process a given string w from right to left. We have shown the naive *ASDAWG*(w) and *MASDAWG*(w) with w = `abba` in Figure 12.1, in which the structures are drawn and arranged for ease of readers to see them in the context of updating *MASDAWG*(u) to *MASDAWG*(au) with $a \in \Sigma^*$ and $u, au \in \mathit{Suffix}(w)$.

Now we consider what happens in constructing *MASDAWG*(au) from *MASDAWG*(u). Due to Lemma 26, we are able to focus only on investigating the relationship between *DAWG*(au) and *DAWG*(u).

Lemma 33 *Let $a \in \Sigma$ and $u \in \Sigma^*$. For any string $x \in \mathit{Substr}(u) - \mathit{Prefix}(au)$, it holds that $\langle au, [x]\rangle \sim_{au} \langle u, [x]\rangle$.*

Proof. $x^{-1}\mathit{Suffix}(au) = x^{-1}(\{au\} \cup \mathit{Suffix}(u)) = x^{-1}\{au\} \cup x^{-1}\mathit{Suffix}(u) = x^{-1}\mathit{Suffix}(u)$, because $x^{-1}\{au\} = \emptyset$ for $x \notin \mathit{Prefix}(au)$. □

The above lemma implies that we have only to care about the prefixes of au in constructing $\mathit{MASDAWG}(au)$ from $\mathit{MASDAWG}(u)$. We need not modify nor change the structure of $\mathit{MASDAWG}(u)$: it is kept static.

Lemma 34 *Let $a \in \Sigma$ and $u \in \Sigma^*$. For any $x \in \mathit{Prefix}(u)$ and $y \in \Sigma^*$, if $\langle au, [ax]\rangle \sim_{au} \langle u, [y]\rangle$ then $[x]^R_u = [y]^R_u$.*

Proof. Since $x \in \mathit{Prefix}(u)$, there exists $s \in \Sigma^*$ such that $u = xs$. By the assumption, $(ax)^{-1}\mathit{Suffix}(au) = y^{-1}\mathit{Suffix}(u)$. Since s is included in the left set, s is also included in the right set, i.e. $s \in y^{-1}\mathit{Suffix}(u)$, which implies $ys \in \mathit{Suffix}(xs)$, thus $y \in \mathit{Suffix}(x)$. We have two cases according to $x \in \mathit{Prefix}(au)$.

(Case 1) When $x \in \mathit{Prefix}(au)$. Since $x \in \mathit{Prefix}(axs)$, $x = a^i$ and $y = a^j$ for some integers $j \leq i$. Suppose $j < i$, and let $k = i - j > 0$. Then $a^k s \in y^{-1}\mathit{Suffix}(u)$ while $a^k s \notin (ax)^{-1}\mathit{Suffix}(au)$, that contradicts with the assumption that $(ax)^{-1}\mathit{Suffix}(au) = y^{-1}\mathit{Suffix}(u)$. Thus $j = i$, which yields $y = x = a^i$.

(Case 2) When $x \notin \mathit{Prefix}(au)$.

$$\begin{array}{rcll} y^{-1}\mathit{Suffix}(u) & = & (ax)^{-1}\mathit{Suffix}(au) & \text{by the assumption} \\ & \subseteq & x^{-1}\mathit{Suffix}(au) & \text{since } x \in \mathit{Suffix}(ax) \\ & = & x^{-1}\mathit{Suffix}(u) & \text{since } x \notin \mathit{Prefix}(au) \\ & \subseteq & y^{-1}\mathit{Suffix}(u) & \text{since } y \in \mathit{Suffix}(x) \end{array}$$

Thus we have $x^{-1}\mathit{Suffix}(u) = y^{-1}\mathit{Suffix}(u)$, that is, $[x]^R_u = [y]^R_u$. □

The path in $\mathit{MASDAWG}(u)$ spelling out u is called its 'backbone'. The above lemma shows that if a node $\langle au, [ax]\rangle$ on the 'backbone' of $\mathit{MASDAWG}(au)$ is equivalent to a node of $\mathit{MASDAWG}(u)$, the node $\langle au, [ax]\rangle$ is also on the 'backbone' of $\mathit{MASDAWG}(u)$. This fact is crucial in order that our algorithm, which will be given in the sequel, performs in time linear in the size of $\mathit{MASDAWG}(u)$.

For the prefixes of string au, we have the following lemma.

Lemma 35 *Let $a \in \Sigma$ and $u \in \Sigma^*$. Let $ax \in \mathit{Prefix}(au)$ be the shortest string which satisfies $\langle au, [ax] \rangle \sim_{au} \langle u, [x] \rangle$. Then for any longer prefix $axv \in \mathit{Prefix}(au)$, it holds that $\langle au, [axv] \rangle \sim_{au} \langle u, [xv] \rangle$.*

Proof. Since $\langle au, [ax] \rangle \sim_{au} \langle u, [x] \rangle$, $(ax)^{-1}\mathit{Suffix}(au) = x^{-1}\mathit{Suffix}(u)$. Therefore, we have $(axv)^{-1}\mathit{Suffix}(au) = v^{-1}((ax)^{-1}\mathit{Suffix}(au)) = v^{-1}(x^{-1}\mathit{Suffix}(u)) = (xv)^{-1}\mathit{Suffix}(u)$. □

Remark that the node $\langle u, [xv] \rangle$ already exists in $\mathit{MASDAWG}(u)$, since $xv \in \mathit{Prefix}(u)$. The above lemma guarantees that all nodes we have to newly create in $\mathit{MASDAWG}(au)$ are $\langle au, [t] \rangle$ for strings $t \in \mathit{Prefix}(z)$, where z is the longest prefix of au which does *not* satisfy $\langle au, [ax] \rangle \sim_{au} \langle u, [x] \rangle$.

Now the next question is how to *efficiently* check whether $\langle au, [ax] \rangle \sim_{au} \langle u, [x] \rangle$ or not for each $x \in \mathit{Prefix}(u)$. Our idea is to count the cardinality of the set $x^{-1}\mathit{Suffix}(u)$.

Lemma 36 *Let $a \in \Sigma$ and $u \in \Sigma^*$. For any $x \in \mathit{Substr}(u)$, $\langle au, [ax] \rangle \sim_{au} \langle u, [x] \rangle$ if and only if $|(ax)^{-1}\mathit{Suffix}(au)| = |x^{-1}\mathit{Suffix}(u)|$.*

Proof. We first show that $(ax)^{-1}\mathit{Suffix}(au) \subseteq x^{-1}\mathit{Suffix}(u)$. Let us choose $s \in (ax)^{-1}\mathit{Suffix}(au)$ arbitrarily. Then $axs \in \mathit{Suffix}(au) = \{au\} \cup \mathit{Suffix}(u)$. If $axs = au$, then $xs = u$. Otherwise, $axs \in \mathit{Suffix}(u)$. Since xs is a suffix of axs, we know that xs is also a suffix of u. In both cases, we have $xs \in \mathit{Suffix}(u)$, which implies that $s \in x^{-1}\mathit{Suffix}(u)$. Thus $(ax)^{-1}\mathit{Suffix}(au) \subseteq x^{-1}\mathit{Suffix}(u)$. It yields that $(ax)^{-1}\mathit{Suffix}(au) = x^{-1}\mathit{Suffix}(u)$ if and only if $|(ax)^{-1}\mathit{Suffix}(au)| = |x^{-1}\mathit{Suffix}(u)|$. By the definition of $\sim_{au}$, we have proved the lemma. □

We associate each node $\langle u, [x] \rangle$ with the cardinality of the set, $|x^{-1}\mathit{Suffix}(u)|$, denoted by $\#\langle u, [x] \rangle$. Note that $\#\langle u, [u] \rangle = 1$ since $u^{-1}\mathit{Suffix}(u) = \{\varepsilon\}$, and that $\#\langle u, [\varepsilon] \rangle = |u| + 1$ since $\varepsilon^{-1}\mathit{Suffix}(u) = \mathit{Suffix}(u)$.

Lemma 37 *Let $a \in \Sigma$ and $u \in \Sigma^*$. For any $x \in \mathit{Prefix}(u)$, $\#\langle au, [ax] \rangle = \#\langle u, [ax] \rangle + 1$.*

Proof. Since $x \in \mathit{Prefix}(u)$, $\#\langle au, [ax] \rangle = |(ax)^{-1}\mathit{Suffix}(au)| = |(ax)^{-1}(\{au\} \cup \mathit{Suffix}(u))| = |(ax)^{-1}\{au\} \cup (ax)^{-1}\mathit{Suffix}(u))| = \#\langle u, [ax] \rangle + 1$. □

The whole algorithm is shown in Figure 12.2. Since the algorithm manipulates an input string w from right to left, we number the characters in w as $w = w_n w_{n-1} \ldots w_1$.

```
Algorithm Construction of $MASDAWG(w = w_n w_{n-1} \dots w_1)$.
 1  create new nodes $s_0$;
 2  $\#(s_0) := 1$;   $\#(\mathbf{nil}) := 0$;
 3  $initNode[0] := s_0$;  $node := s_0$;
 4  for $i := 1$ to $n$ do
 5      $s :=$ FIND($node, w_i$);
 6      $target :=$ NEWTARGETNODE($s, i-1, node$);
 7      $newNode :=$ create a new node with copying all out-going edges of $node$;
 8      add or overwrite edge ($newNode, w_i, target$);
 9      $\#(newNode) := i$;
10      $initNode[i] = newNode$;
11      $node = newNode$;

function NEWTARGETNODE(Node $s$, int $j$, Node $backbone$) : Node
 1  $nextNumSuf := \#(s) + 1$;
 2  if $nextNumSuf = \#(backbone)$ then return $backbone$;    /* redirection */
 3  $nextBackbone :=$ FIND($backbone, w_j$);
 4  $newNode :=$ create a new node with copying all out-going edges of $s$;
 5  $s :=$ FIND($s, w_j$);
 6  $target :=$ NEWTARGETNODE($s, j-1, nextBackbone$);
 7  add or overwrite edge ($newNode, w_j, target$);
 8  $\#(newNode) := nextNumSuf$;
 9  return $newNode$;

function FIND(Node $s$, char $c$) : Node
 1  if $s$ has the $c$-edge then
 2      let $(s, c, r)$ be the $c$-edge from $s$;
 3      return $r$;
 4  else return nil;
```

Figure 12.2: The algorithm to construct $MASDAWG(w)$.

When we read w from right to left, a substring $w_i \dots w_j$ of w is represented by $w_{[i:j]}$. Note that $i \geq j$ in this case. An edge is represented by a triple (r, w_i, s), where s, r are nodes and w_i is the character for the label of the edge.

Theorem 26 *For any string $w \in \Sigma^*$, our algorithm constructs $MASDAWG(w)$ in time linear in its size.*

Proof. In the i-th phase of the main routine, $MASCDAWG(w_{[i:]})$ is incrementally constructed based on $MASCDAWG(w_{[i-1:]})$. Remark that in any call of the function

NewTargetNode, the following pre-conditions are satisfied.

$$\begin{array}{lcl} \mathit{backbone} & = & \langle w_{[i-1:]}, [w_{[i-1:j]}]\rangle, \\ s & = & \begin{cases} \langle w_{[i-1:]}, [w_{[i:j]}]\rangle & \text{if } w_{[i:j]} \in \mathit{Substr}(w_{[i-1:]}), \\ \mathbf{nil} & \text{otherwise.} \end{cases} \end{array}$$

The variable *backbone* expresses the j-th node on the backbone of $\mathit{MASCDAWG}(w_{[i-1:]})$ from the initial node. In line 12.1 in NewTargetNode, the function Find never returns **nil** because *backbone* has w_j-edge. On the other hand, the variable s represents the node, called the referenced node, in $\mathit{MASCDAWG}(w_{[i-1:]})$ which corresponds to the prefix $w_{[i:j]}$ of the string $w_{[i:]}$. The basic role of NewTargetNode is to create a new node *newNode*, that is a copy of the referenced node s except the only one edge along the prefix $w_{[i:j]}$. Lemma 33 guarantees that other edges are unchanged. However, if a new node becomes equivalent to an existing node in $\mathit{MASCDAWG}(w_{[i-1:]})$, we have to redirect the edge instead of creating it. Thanks to Lemma 34, we do not have to examine all nodes in $\mathit{MASCDAWG}(w_{[i-1:]})$. Candidates are always on the backbone of $\mathit{MASCDAWG}(w_{[i-1:]})$, and in fact the only possible candidate is pointed by the variable *backbone*. By Lemma 36, checking the equivalence is performed by merely comparing the cardinality of the sets, stored by $\#(\cdot)$ in the pseudo-code. Moreover, the cardinality of a new node is simply computed by $\#(s) + 1$ due to Lemma 37. Once an equivalent node is found among the existing nodes of $\mathit{MASCDAWG}(w_{[i-1:]})$, we can immediately terminate the recursive calls, since Lemma 35 guarantees that the rest of the new backbone will be equivalent to the current one. Since $\#(\mathbf{nil}) = 0$ and $\#(s_0) = 1$, the recursive call never falls into infinite loop.

At each call of NewTargetNode except the last one, a new node is created. Thus the running time of the algorithm is linear with respect to the output size. □

The on-line (right-to-left) construction of $\mathit{MASDAWG}(w)$ where $w = \texttt{abaa\$}$ is displayed in Figure 12.3.

12.2 Minimum All-Suffixes Compact Directed Acyclic Word Graphs

To achieve a more space-economical index structure for all suffixes of a string, we turn our attention to compact directed acyclic word graphs (CDAWGs) and consider their

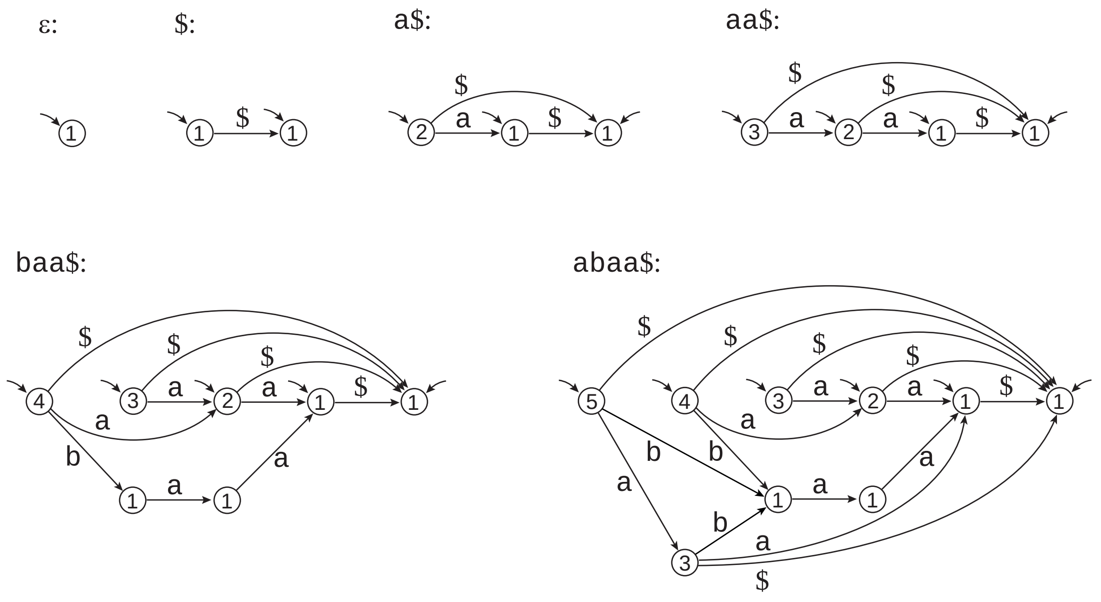


Figure 12.3: Construction of *MASDAWG*(abaa\$). Each node is marked by $\#\langle u, [x]\rangle$ where $u =$ abaa\$ and $x \in \mathit{Substr}(u)$.

all-suffixes version. Recall Definition 13 for *CDAWG*(w).

Lemma 38 *Let $x \in \mathit{Substr}(w)$. Assume $\overset{w}{\overrightarrow{x}} \notin \mathit{Suffix}(w)$. Then, x occurs in w at least twice.*

Proof. For a contradiction, assume x occurs in w only once. We have $|\mathit{Prefix}(w)x^{-1}| = 1$. Let $w = hxy$. Since x occurs in w only once, $|\mathit{Prefix}(w)x^{-1}| = |\mathit{Prefix}(w)(xy)^{-1}|$. Thus $x \equiv_w^L xy$ and $\overset{w}{\overrightarrow{x}} = xy$. However, $xy \in \mathit{Suffix}(w)$, a contradiction. Consequently, x appears in w at least twice. □

The following corollary derives from Lemma 38.

Corollary 5 *Assume that w terminates with a unique symbol \$. Then, for any string $x \in \mathit{Substr}(w) - \mathit{Suffix}(w)$, node $[\overset{w}{\overrightarrow{x}}]_w^R$ is of out-degree more than one.*

Definition 34 *ASCDAWG*(w) is a kind of dag with $|w| + 1$ initial nodes, designated by $0, 1, \ldots, |w|$, in which the subgraph consisting of the nodes reachable from the k-th initial node and their out-going edges is *CDAWG*($w[k + 1 :]$).

The simple collection of *CDAWG*($w[1 :]$), *CDAWG*($w[2 :]$),…, *CDAWG*($w[n]$), *CDAWG* ($w[n + 1 :]$) ($n = |w|$) is an example of *ASCDAWG*(w), which is referred to as the *naive ASCDAWG*(w). The number of nodes of the naive *ASCDAWG*(w) is $O(|w|^2)$, simply because the number of nodes of *CDAWG*(w) is $O(|w|)$. We now introduce the minimized version of *ASCDAWG*(w), which is well defined similarly to $MASDAWG(w)$. Each node of *ASCDAWG*(w) can be represented by a pair $\langle u, [\overset{u}{\overrightarrow{x}}]^R_u\rangle$ with $u \in \mathit{Suffix}(w)$ and $x \in \mathit{Substr}(u)$. We write $\langle u, [\overset{u}{\overrightarrow{x}}]^R_u\rangle$ simply as $\langle u, [\overrightarrow{x}]\rangle$ when no confusion occurs. If $\langle u, [\overset{u}{\overrightarrow{x}}]^R_u\rangle \sim_w \langle v, [\overset{v}{\overrightarrow{y}}]^R_v\rangle$, we merge these nodes and the resulting structure is the *minimum ASCDAWG*(w), denoted by *MASCDAWG*(w). Obviously, *MASCDAWG*(w) can be obtained by the DAG-minimization algorithm [60] that runs in time linear in the number of edges of the naive *ASCDAWG*(w). As for the size of MASCDAWGs, we have the following:

Theorem 27 *When* $|\Sigma| \geq 2$*, the number of nodes in MASCDAWG*(w) *for a string* w *is* $\Theta(|w|^2)$*. It is* $\Theta(|w|)$ *for a unary alphabet.*

In the following, we give some preliminaries for our algorithm for construction of MASCDAWGs. The algorithm also processes a given string from right to left. That is, it updates *MASCDAWG*(u) to *MASCDAWG*(au) with $a \in \Sigma$ and $u \in \Sigma^*$. Here, we have only to consider a string $x \in \mathit{Substr}(w)$ such that $\overset{w}{\overrightarrow{x}} = x$. Since Proposition 16 and Lemma 26 hold for an arbitrary string in *Substr*(w), it is guaranteed that the CDAWGs sharing a node in *MASCDAWG*(w) are also 'consecutive'. Therefore, we only consider the relationship between *CDAWG*(au) and *CDAWG*(u), two consecutive CDAWGs.

Lemma 39 *Let* $a \in \Sigma$ *and* $u \in \Sigma^*$*. For any string* $x \in \mathit{Substr}(u) - \mathit{Prefix}(au)$*,* $\overset{u}{\overrightarrow{x}} = \overset{au}{\overrightarrow{x}}$*.*

Proof. Since $x \notin \mathit{Prefix}(au)$, there is no new occurrence of x in au. It implies that $a(\mathit{Prefix}(u)x^{-1}) = \mathit{Prefix}(au)x^{-1}$. Thus we have $[x]^L_u = [x]^L_{au}$. Consequently, $\overset{u}{\overrightarrow{x}} = \overset{au}{\overrightarrow{x}}$. □

The above lemma ensures that any implicit node of *CDAWG*(u) does not become explicit in *CDAWG*(au) if it is not associated with a prefix of au. It follows from this lemma and Lemma 33 that we do not need to modify nor change the structure of *MASCDAWG*(u) when constructing *MASCDAWG*(au).

Lemma 40 *Let* $a \in \Sigma$ *and* $u \in \Sigma^*$*. For any* $x, z \in \mathit{Substr}(u)$*, if* $\overset{au}{\overrightarrow{ax}} = az$ *then* $\overset{u}{\overrightarrow{z}} = z$*.*

Proof. Suppose contrarily that $\overset{u}{\overrightarrow{z}} \neq z$. That means there exists $y \in \Sigma^*$ such that $\mathit{Prefix}(u)y^{-1} = \mathit{Prefix}(u)z^{-1}$ and $|y| > |z|$. Then $\mathit{Prefix}(au)(ay)^{-1} = (\mathit{Prefix}(au)y^{-1})a^{-1} = (a(\mathit{Prefix}(u)y^{-1}))a^{-1} = (a(\mathit{Prefix}(u)z^{-1}))a^{-1} = \mathit{Prefix}(au)(az)^{-1} = \mathit{Prefix}(au)(ax)^{-1}$. Thus $ay \equiv^L_{au} ax$ and $|ay| > |az|$. It contradicts the assumption $\overset{au}{\overrightarrow{ax}} = az$. □

Lemma 41 *Let $a \in \Sigma$ and $u \in \Sigma^*$. For any $x \in Prefix(u)$ and $y \in \Sigma^*$ satisfying $\langle au, [\overset{au}{\overrightarrow{ax}}]^R_{au}\rangle \sim_{au} \langle u, [\overset{u}{\overrightarrow{y}}]^R_u\rangle$, there exists $z \in Prefix(u)$ such that $[\overset{u}{\overrightarrow{z}}]^R_u = [\overset{u}{\overrightarrow{y}}]^R_u$.*

Proof. Let z be the string with $\overset{au}{\overrightarrow{ax}} = az$. Then we have $\overset{u}{\overrightarrow{z}} = z$ by Lemma 40. Moreover, $z \in Prefix(u)$ since $x \in Prefix(u)$. Since $\langle au, [az]^R_{au}\rangle = \langle au, [\overset{au}{\overrightarrow{ax}}]^R_{au}\rangle \sim_{au} \langle u, [\overset{u}{\overrightarrow{y}}]^R_u\rangle$, we have $[z]^R_u = [\overset{u}{\overrightarrow{y}}]^R_u$ by Lemma 34. Thus $[\overset{u}{\overrightarrow{z}}]^R_u = [\overset{u}{\overrightarrow{y}}]^R_u$. □

Lemma 41 shows that if node $\langle au, [\overset{au}{\overrightarrow{ax}}]^R_{au}\rangle$ on the 'backbone' of *MASCDAWG*(au) is equivalent to a node of *MASCDAWG*(u), the node $\langle au, [\overset{au}{\overrightarrow{ax}}]^R_{au}\rangle$ is also on the 'backbone' of *MASCDAWG*(u). It corresponds to Lemma 34.

We have the following lemma which corresponds to Lemma 35.

Lemma 42 *Let $a \in \Sigma$ and $u \in \Sigma^*$. Let $ax \in Prefix(au)$. Let $\overset{au}{\overrightarrow{ax}}$ be the shortest string for which there exists $z \in Prefix(u)$ such that $\langle au, [\overset{au}{\overrightarrow{ax}}]^R_{au}\rangle \sim_{au} \langle u, [\overset{u}{\overrightarrow{z}}]^R_u\rangle$. Let $\overset{au}{\overrightarrow{ax}} = ay$. Then for any longer prefix $ayv \in Prefix(au)$, there exists $s \in Prefix(u)$ such that $\langle au, [\overset{au}{\overrightarrow{ayv}}]^R_{au}\rangle \sim_{au} \langle u, [\overset{u}{\overrightarrow{s}}]^R_u\rangle$.*

Proof. Let $\overset{au}{\overrightarrow{ayv}} = as$. By Lemma 40, $\overset{u}{\overrightarrow{s}} = s$. Since $yv \in Prefix(u)$, $s \in Prefix(u)$. Let $\overset{u}{\overrightarrow{z}} = t$. By the assumption $\langle au, [\overset{au}{\overrightarrow{ax}}]^R_{au}\rangle \sim_{au} \langle u, [\overset{u}{\overrightarrow{z}}]^R_u\rangle$, we have $\langle au, [ay]\rangle \sim_{au} \langle u, [t]\rangle$. Since $y \in Prefix(u)$, $\langle au, [ay]\rangle \sim_{au} \langle u, [y]\rangle$ by Lemma 34. Note that $y \in Prefix(s)$. Hence we have $\langle au, [as]\rangle \sim_{au} \langle u, [s]\rangle$ by Lemma 35. Because $as = \overset{au}{\overrightarrow{ayv}}$ and $s = \overset{u}{\overrightarrow{s}}$, it holds that $\langle au, [\overset{au}{\overrightarrow{ayv}}]^R_{au}\rangle \sim_{au} \langle u, [\overset{u}{\overrightarrow{s}}]^R_u\rangle$. □

We remark that the equivalence $\langle au, [\overset{au}{\overrightarrow{ax}}]^R_{au}\rangle \sim_{au} \langle u, [\overset{u}{\overrightarrow{z}}]^R_u\rangle$ can also be examined by checking the cardinalities of the corresponding sets, as is the case of MASDAWGs. Hereby we have shown that *MASCDAWG*(w) can be constructed in a similar way to *MASDAWG*(w). The only thing not clarified yet is whether or not *MASCDAWG*(w) can be built in time linear in its size. We establish the following lemmas to support the linearity.

Lemma 43 *Let $a \in \Sigma$ and $w \in \Sigma^*$. For any $x, z \in Substr(w)$, if $\overset{w}{\overrightarrow{ax}} = az$ then $\overset{w}{\overrightarrow{z}} = z$.*

Proof. For a contradiction, assume $\overset{w}{\overrightarrow{z}} \neq z$. Then there exists $y \in \Sigma^*$ such that $Prefix(w)y^{-1} = Prefix(w)z^{-1}$ and $|y| > |z|$. Then $Prefix(w)(ay)^{-1} = (Prefix(w)y^{-1})a^{-1} = (Prefix(w)z^{-1})a^{-1} = Prefix(w)(az)^{-1}$. Thus $ay \equiv^L_{au} az$ and $|ay| > |az|$. It contradicts the assumption $\overset{w}{\overrightarrow{ax}} = az$. □

Note that the statement of the above lemma slightly differs from that of Lemma 40.

Lemma 44 *Let $a, b \in \Sigma$ and $w \in \Sigma^*$. Let $x, y \in Substr(w)$ such that $\overset{w}{\overrightarrow{xb}} = xby \neq w$. If $axb \in Substr(w)$, then $axby \in Substr(w)$, and $\overset{w}{\overrightarrow{axby'}} = \overset{w}{\overrightarrow{axby}}$ for any $y' \in Prefix(y)$.*

Proof. Since $axb \in Substr(w)$ and $xby \neq w$, there always exists $z \in \Sigma^*$ such that $\overset{w}{\overrightarrow{axb}} = axbz \in Substr(w)$. By Lemma 43, $\overset{w}{\overrightarrow{xbz}} = xbz$. Since $\overset{w}{\overrightarrow{xb}} = xby$, $y \in Prefix(z)$. Because $axbz \in Substr(w)$, $axby \in Substr(w)$. For any $y' \in Prefix(y)$, $axbz \equiv^L_w axby'$ since $\overset{w}{\overrightarrow{axb}} = axbz$. Therefore $\overset{w}{\overrightarrow{abxy'}} = abxz = \overset{w}{\overrightarrow{abxy}}$. □

Suppose $\overset{w}{\overrightarrow{x}} = x$. If we in advance know node $[\overset{w}{\overrightarrow{x}}]^R_w$ has an out-going edge labeled with by, we can avoid to scan the whole string xby in traversing the path $axby$ from the initial node of $CDAWG(w)$. Moreover, it is guaranteed that the path by from the (explicit or implicit) node for ax consists of one edge: no explicit node is contained in the path. This is a key to achieve an algorithm that constructs $MASCDAWG(w)$ in linear time with respect to its size.

The whole algorithm is shown in Figure 12.4. Here we also read an input string w from right to left, and thus w is written as $w = w_n w_{n-1} \dots w_1$. The label $w_i w_{i-1} \dots w_j$ of each edge can be represented by a pair of the beginning position i and the ending position $j - 1$. $(i > j - 1)$ If the string corresponding to the label appears in w more than once, we represent it by the leftmost occurrence. This way we can assign $endpos(s)$ to a node s, where $endpos(s)$ indicates the ending position of every in-coming edge of s. Thereby, we represent each edge by a triple (r, i, s), where r, s are explicit nodes. An implicit node corresponding to some substring x of w can be represented by a triple (r, k, p), where r is an explicit parent node of the implicit node. Assuming the representative of the equivalence class associated with r is y, $x = yu$ where $u = w_k w_{k-1} \dots w_p$. The quartet (r, k, p, s) is called the *reference quartet*, where s is the closest explicit child node of r reachable via the w_k-edge from r. When $|p - k|$ is minimum, the quartet (r, k, p, s) is called the *canonical reference quartet*.

Theorem 28 *For any string $w \in \Sigma^*$, our algorithm constructs $MASCDAWG(w)$ in time linear in its size.*

```
Algorithm Construction of $MASCDAWG(w = w_n w_{n-1} \ldots w_1)$.
 1  create new nodes $s_0$,$s_1$,$s_2$;
 2  $\#(s_0) := 1$;  $\#(s_1) := 1$;  $\#(s_2) := 2$;  $\#(\mathbf{nil}) := 0$;
 3  $endpos(s_0) := 0$;  $endpos(s_1) := 1$;  $endpos(s_2) := 2$;  $endpos(\mathbf{nil}) := 0$;
 4  add edges $(s_1, 1, s_0)$, $(s_2, 1, s_0)$, $(s_2, 2, s_0)$;
 5  $initNode[0] := s_0$;  $initNode[1] := s_1$;  $initNode[2] := s_2$;  $node := s_2$;
 6  for $i := 3$ to $n$ do
 7      $(s, k, p, r) :=$ CANONIZE(FASTFIND($node, i, 1$));
 8      $target :=$ NEWTARGETNODE($(s, k, p, r), i-1, node$);
 9      $newNode :=$ create a new node with copying all out-going edges of $node$;
10      add or overwrite edge $(newNode, i, target)$;
11      $\#(newNode) := i$;    $endpos(newNode) := i$;
12      $initNode[i] = newNode$;    $node = newNode$;

function NEWTARGETNODE(refQuartet $(s, k, p, r)$, int $j$, Node $backbone$) : Node
 1  $nextNumSuf := \#(r) + 1$;
 2  if $nextNumSuf = \#(backbone)$ then return $backbone$;    /* redirection */
 3  let $(backbone, \ell, nextBackbone)$ be the $w_j$-edge from $backbone$;
 4  $m := \ell - endpos(nextBackbone)$;    /* length of this edge */
 5  if $k = p$ then    /* explicit node */
 6      $newNode :=$ create a new node with copying all out-going edges of $s$;
 7      $(s, k, p, r) :=$ CANONIZE(FASTFIND($s, j, m$));
 8      $target :=$ NEWTARGETNODE($(s, k, p, r), j-m, nextBackbone$);
 9      add or overwrite edge $(newNode, j, target)$;
10      $\#(newNode) := nextNumSuf$;    $endpos(newNode) := j$;
11      return $newNode$;
12  else if $w_p = w_j$ then /* implicit and next characters are the same */
13      $(s, k, p, r) :=$ CANONIZE($s, k, p-m, r$);    /* skip $m$ characters */
14      return  NEWTARGETNODE($(s, k, p, r), j-m, nextBackbone$);
15  else    /* implicit and next characters are different */
16      $newNode :=$ create a new node;    /* edge split */
17      add new edges $(newNode, p, r)$ and $(newNode, j, s_0)$;
18      $\#(newNode) := nextNumSuf$;    $endpos(newNode) := j$;
19      return $newNode$;

function FASTFIND(Node $s$, int $i$, int $length$) : refQuartet
/* compute the position from $s$ along the string $w_i w_{i-1} \ldots w_{i-length+1}$ */
/* remark that $\underline{\text{the first character } w_i \text{ is only compared}}$ */
 1  if $s$ has the $w_i$-edge then
 2      let $(s, \ell, r)$ be the $w_i$-edge from $s$;
 3      return $(s, \ell, \ell - length, r)$;
 4  else return $(s, i, i - length, \mathbf{nil})$;

function CANONIZE( refQuartet $(s, k, p, r)$ ) : refQuartet
/* when the referenced position is an explicit node, canonize the expression */
 1  if $k > p$ and  $p = endpos(r)$ then return $(r, p, p, r)$;
 2  else return $(s, k, p, r)$;
```

Figure 12.4: The algorithm to construct $MASCDAWG(w)$.

Proof. Firstly, remark that in any call of NewTargetNode, the following preconditions are satisfied.

$$\begin{array}{lcl} \mathit{backbone} & = & \langle w_{[i-1:]}, [\overrightarrow{w_{[i-1:j]}}] \rangle, \\ s & = & \langle w_{[i-1:]}, [\overrightarrow{y}] \rangle, \\ v & = & w_{[k:p]}, \\ r & = & \begin{cases} \langle w_{[i-1:]}, [\overrightarrow{x}] \rangle & \text{if } x = w_{[i:j]}, \\ \textbf{nil} & \text{otherwise}, \end{cases} \end{array}$$

where x is the longest string in $\mathit{Prefix}(w_{[i:j]}) \cap \mathit{Substr}(w_{[i-1:]})$, y is the longest prefix of x satisfying $\overset{w_{[i-1:]}}{\overrightarrow{y}} = y$, and v is the string such that $yv = w_{[i-1:j]}$. It is important to notice that the reference quartet (s, k, p, r) is a generalization of the reference node s in a MASDAWG. It can treat implicit nodes as well as explicit nodes. The reference quartet (s, k, p, r) represents an explicit node if and only if $k = p$.

The basic structure of the algorithm is similar to that for MASDAWGs. A big difference is that the referenced node may be *implicit*, while *backbone* is always *explicit* and *backbone* always has the w_j-edge. Lemmas 39, 41, and 42 fill the gap in showing the correctness.

A subtle point is that in function FastFind, we compare only the first character even when traversing a string of *length* ≥ 2. Lemma 44 guarantees its correctness. The lemma also gives the validity of line 12.2 in function NewTargetNode, where we skip m characters whenever the first characters w_p and w_j are the same.

We now verify the running time. Note that FastFind takes constant time regardless the *length*, because it only compare the first character. In the i-th phase of the algorithm, *backbone* traverses the first portion of the backbone in $MASCDAWG(w_{[i-1:]})$. In each call of NewTargetNode, the value of *backbone* is changed to the next node on the backbone. Unfortunately, however, it is not enough to guarantee the linearity of the algorithm. A very delicate point is that in lines 12.2 and 12.2 of NewTargetNode, *backbone* proceeds without creating a new node! To overcome this difficulty, let us remark that in the next phase, the new backbone consists of the sequence of the nodes created in the last phase, followed by the rest portion of the last backbone. This means that every node (except the unique sink node) in MASCDAWG will be touched in NewTargetNode at most once. Thus the total running time is linear with respect to the size, the number of nodes in the resulting MASCDAWG. □

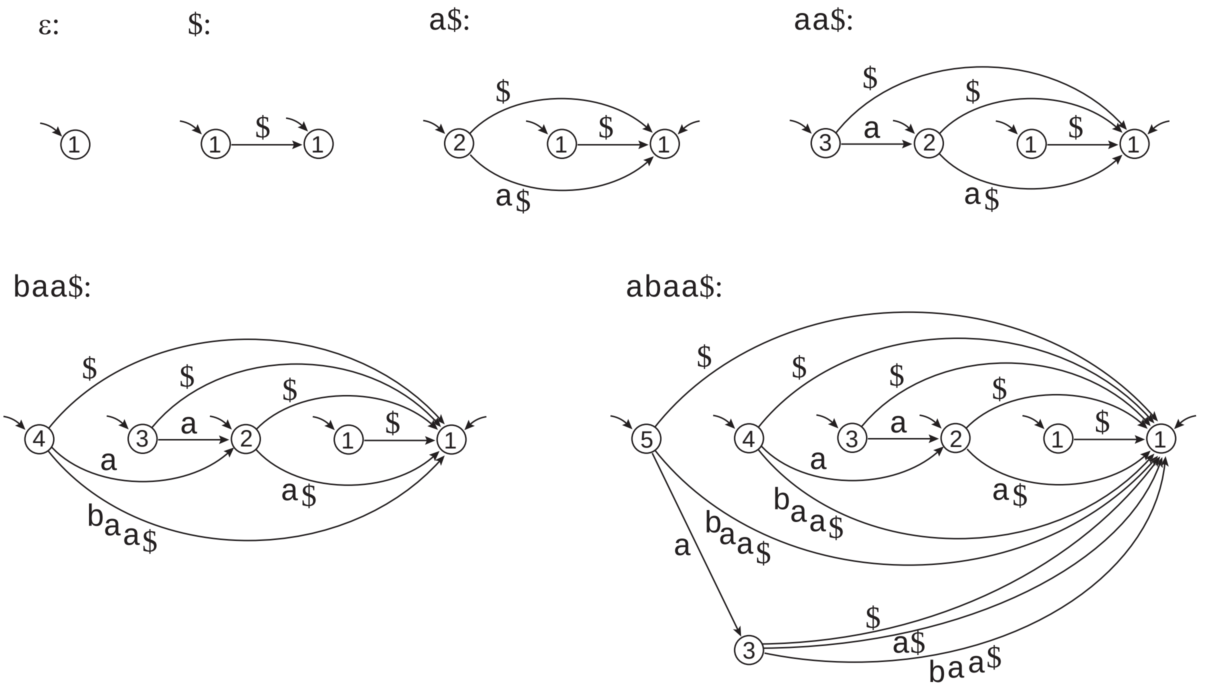


Figure 12.5: Construction of *MASCDAWG*(abaa\$).

The on-line (right-to-left) construction of *MASCDAWG*(w) where w = abaa\$ is displayed in Figure 12.5.

12.3 Concluding Remarks

It is easy to construct the *minimum all-suffixes suffix trie* in time proportional to its size, by a slightly modified algorithm for the MASDAWG. We only need to care not to merge subtrees of the same suffix trie, so that the resulting structure does *not* become a dag. Similarly, the *minimum all-suffixes suffix tree* can also be built in time linear to its size, by modifying the algorithm for the MASCDAWG.

Chapter 13

Pattern Discovery from String Data Sets

A vast amount of data is available today, and discovering useful *rules* from those data is quite important. Very commonly, information is stored and manipulated as *strings*. In the context of strings, rules are *patterns*. Given two sets of strings, often referred to as *positive examples* and *negative examples*, it is desired to find the pattern that is the most common to the former and the least common to the latter. This task of finding the best pattern in the sense of separating two given sets of strings is critical to discovery science as well as machine learning. In fact, pattern discoveries from genomical sequence data will give us a good knowledge to characterize the data.

Shimozono et al. developed knowledge discovery system BONSAI [63] that outputs a decision tree based on the best *substring patterns* for separating two input data sets $S, T \in \Sigma^*$. The best substring pattern separating S and T can be found in linear time by a clever use [32] of the suffix tree for $S \cup T$. In order that BONSAI system can deal with *subsequence patterns* as well, Hirao et al. [28] proposed a practical algorithm to find the subsequence pattern that is the most abundant in one set and the rarest in the other. Since this problem is NP-hard, they employed efficient pruning heuristics to reduce the number of candidate patters for the best subsequence. Also, the matching phase of their algorithm was significantly sped up by means of directed acyclic subsequence graphs (DASGs) recalled in Section 10.1. The actually efficiency of this algorithm was reported in [27].

In this chapter, we firstly present a practical algorithm to discover the best *episode patterns* to separate two given sets of strings. An episode pattern is a pair $\langle p, k \rangle$, where

p is a string and k is an integer. It said to *match* a string w if p is a subsequence of a substring u of w with $|u| \leq k$ [54, 19]. We stress that episode patterns are generalized concept of substring patterns as well as subsequence patterns, since the substring pattern matching with $p \in \Sigma^*$ corresponds exactly to the episode pattern matching with $\langle p, |p| \rangle$, and the subsequence pattern matching with p is the same as the episode pattern matching with $\langle p, \infty \rangle$. We remark that the problem of finding the best episode patterns is also NP-hard. As well as the case of subsequence patterns, we employ similar pruning heuristics based on the properties of episode patterns and the evaluation function calculating the scores of patterns. We make the matching phase of our algorithm fast by using *episode directed acyclic word graphs* (*EDASGs*) proposed in [70]. Bannai et al. [7] and Iida et al. [33] installed this algorithm into the BONSAI system and evaluated its efficiency.

Secondly, we consider *VLDC patterns* for a pattern class for the purpose of separating two given sets of strings. VLDC patterns can be seen as *regular patterns* [65] in which the variable $\star$ is allowed to be substituted with the empty string ε. VLDC patterns are generalization of substring patterns as well as subsequence patterns. For instance, substring pattern matching with $\mathtt{aba} \in \Sigma^*$ corresponds to VLDC pattern matching with $\star\mathtt{abc}\star$, and subsequence pattern matching with $\mathtt{abc}$ is the same as VLDC pattern matching with $\star\mathtt{a}\star\mathtt{b}\star\mathtt{c}\star$. The *language* of a VLDC pattern $q \in \Pi$ is defined to be the set of strings obtained by replacing $\star$'s in q with arbitrary strings in Σ^*. It corresponds to a class of the *pattern languages* proposed by Angluin [3].

Our algorithm for efficient discovery of the best VLDC patterns from two given two sets of strings is also sped up in two ways, namely, by pruning heuristics and fast VLDC pattern matching algorithms. We accelerate the matching phase by the two VLDC pattern matching algorithms introduced in Section 10.3.

We also present a practical algorithm to find the best pair $\langle q, k \rangle$ to distingish two given sets of strings, where $q \in \Pi$ and $k \in \mathcal{N}$. Specifying the length of an occurrence of a VLDC pattern is of great significance especially when classifying *long* strings over a *small* alphabet, since a short VLDC pattern surely matches most long strings. Therefore, for example, when two sets of biological sequences are given to be separated, this approach is adequate and promising.

We declare that this work generalizes and outperforms the ones accomplished in [28], since it is capable of discovering more advanced and useful patterns. In fact, we show some experimental results that convince us of the accuracy of our algorithms as well as their fast performances.

These results were originally published in [29, 64, 35, 8].

13.1 Finding Best Patterns from Sets of Strings

Let *good* be a function from $\Sigma^* \times 2^{\Sigma^*} \times 2^{\Sigma^*}$ to the set of real numbers. The problem we tackle here is defined as follows.

Definition 35 (Finding the best pattern according to *good*)
Input: Two sets S, T of strings.
Output: A pattern p that maximizes the score of $\mathit{good}(p, S, T)$.

Intuitively, the score of $\mathit{good}(p, S, T)$ expresses the "goodness" of p in the sense of distinguishing S from T. The definition of *good* varies with applications. 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 p. Any of the above-mentioned measures can be expressed by the following form:

$$\begin{aligned} \mathit{good}(p, S, T) &= f(x_p, y_p, |S|, |T|), \text{ where} \\ x_p &= |S \cap L(p)|, \\ y_p &= |T \cap L(p)|, \end{aligned}$$

and $L(p) = \{w \in \Sigma^* \mid p \text{ matches } w\}$. For instance, the entropy information gain [59], which is used in the BONSAI system [63], is described in terms of the function *good*, as follows:

$$\begin{aligned} f(x, y, x_{\max}, y_{\max}) &= -\frac{x+y}{x_{\max}+y_{\max}} I(x, y) \\ &\quad -\frac{x_{\max}-x+y_{\max}-y}{x_{\max}+y_{\max}} I(x_{\max}-x, y_{\max}-y), \\ \text{where } I(x, y) &= \begin{cases} 0 & \text{if } x = 0 \text{ or } y = 0 \\ -\frac{x}{x+y}\log\frac{x}{x+y} - \frac{y}{x+y}\log\frac{y}{x+y} & \text{otherwise.} \end{cases} \end{aligned}$$

When S and T are fixed, $x_{\max} = |S|$ and $y_{\max} = |T|$ are regarded as constants. On this assumption, we abbreviate the notation of the function to $f(x, y)$ in the sequel.

In Figure 13.1 we show an exhaustive search algorithm for solving the problem of Definition 35.

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

Figure 13.1: Exhaustive search algorithm.

Since the function $good(p, S, T)$ expresses the goodness of p in the sense of distinguishing the two sets, it is natural to assume that the function f is *conic*, defined as follows:

Definition 36 Function f from $[0, x_{\max}] \times [0, y_{\max}]$ to real numbers is *conic* if

- for any $0 \leq y \leq y_{\max}$, 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 $y_1 \leq y < y' \leq y_{\max}$.

Actually, the χ^2 values, entropy information gain, and gini index are all conic. We remark that every convex fuction is conic. In the sequel, we assume that f is conic and can be evaluated in constant time.

We have the following lemma deriving from the conicality of function f.

Lemma 45 (Hirao et al. [28]) *For any* $0 \leq x < x' \leq x_{\max}$ *and* $0 \leq y < y' \leq y_{\max}$, *we have* $f(x, y) \leq \max\{f(x', y'), f(x', 0), f(0, y'), f(0, 0)\}$.

13.2 Finding Best Substring Patterns

For $p, w \in \Sigma^*$, we write as $p \preceq_{\text{str}} w$ if $p \in Substr(w)$.

Definition 37 Let $p \in \Sigma^*$. The *substring language* of p is defined by

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

The following lemma is quite clear from the above definition.

Lemma 46 (Hirao et al. [28]) *For any $p, u \in \Sigma^*$, if $p \preceq_{\text{str}} u$, then $L^{\text{str}}(p) \supseteq L^{\text{str}}(u)$.*

Definition 38 (Finding the best substring pattern according to f)
Input: Two sets $S, T \subseteq \Sigma^*$ of strings.
Output: A pattern $p \in \Sigma^*$ that maximizes the score of $f(x_p, y_p)$, where $x_p = |S \cap L^{\text{str}}(p)|$ and $y_p = |T \cap L^{\text{str}}(p)|$.

It is stated in [28] that the above problem is solvable in $O(\|S\| + \|T\|)$ time by a clever use [32] of *STree*$(S \cup T)$.

13.3 Finding Best Subsequence Patterns

For $p, w \in \Sigma^*$, we write as $p \preceq_{\text{seq}} w$ if p is a subsequence of w.

Definition 39 Let $p \in \Sigma^*$. The *subsequence language* of p is defined by

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

The following lemma is quite clear from the above definition.

Lemma 47 (Hirao et al. [28]) *For any $p, u \in \Sigma^*$, if $p \preceq_{\text{seq}} u$, then $L^{\text{seq}}(p) \supseteq L^{\text{seq}}(u)$.*

Definition 40 (Finding the best subsequence pattern according to f)
Input: Two sets $S, T \subseteq \Sigma^*$ of strings.
Output: A pattern $p \in \Sigma^*$ that maximizes the score of $f(x_p, y_p)$, where $x_p = |S \cap L^{\text{seq}}(p)|$ and $y_p = |T \cap L^{\text{seq}}(p)|$.

The above problem is known to be NP-hard [28], which implies we have faced exponentially many candidates for the best pattern p. To solve this problem quickly in practice, therefore, we take the following two strategies:

1. Restrict the number of candidate patterns in line 2 of Figure 13.1,

2. Speed up the matching phase in lines 3 and 4 of Figure 13.1.

To do the first one, Hirao et al. [28] used efficient pruning heuristics inspired by Morishita and Sese [58]. The pruning heuristics is based on the following lemma, which derives from Lemma 45 and Lemma 47.

Lemma 48 (Hirao et al. [28]) *For any strings $p, u \in \Sigma^*$, if $p \preceq_{\text{seq}} u$, then $f(x_u, y_u) \leq \max\{f(x_p, y_p), f(x_p, 0), f(0, y_p), f(0, 0)\}$.*

The following problem corresponds to our second strategy given above.

Definition 41 (Counting the matched subsequence patterns)
Input: A set $S \subseteq \Sigma^*$ of strings.
Query: A pattern $p \in \Sigma^*$.
Output: The cardinality of set $S \cap L^{\text{seq}}(p)$.

This is a sub-problem of the one in Definition 40. It has to be answered as quickly as possible, since we are given quite many patterns as queries. To solve this problem, Hirao et al. [28] suggested to construct $DASG(s)$ for each $s \in S$. Then, answering the above query takes $O(\|S\|)$ preprocessing time and $O(|S| \cdot |p|)$ running time.

The whole algorithm to find the best subsequence pattern from two given sets of strings according to function f, proposed by Hirao et al. [28], is shown in Figure 13.2.

13.4 Finding Best Episode Patterns

Definition 42 Let $p \in \Sigma^*$ and $k \in \mathcal{N}$. The *episode language* of $\langle p, k\rangle$ is defined by

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

We have the following lemma according to the above definition.

Lemma 49 *For any $\langle p, k\rangle$ and $\langle u, j\rangle$ with $p, u \in \Sigma^*$ and $k, j \in \mathcal{N}$, if $p \preceq_{\text{seq}} u$ and $k \leq j$, then $L^{\text{eps}}(\langle p, k\rangle) \supseteq L^{\text{eps}}(\langle u, j\rangle)$.*

Definition 43 (Finding the best episode pattern according to f)
Input: Two sets $S, T \subseteq \Sigma^*$ of strings.
Output: A pair $\langle p, k\rangle$ that maximizes the score of $f(x_{\langle p,k\rangle}, y_{\langle p,k\rangle})$, where $x_{\langle p,k\rangle} = |S \cap L^{\text{eps}}(\langle p, k\rangle)|$ and $y_{\langle p,k\rangle} = |T \cap L^{\text{eps}}(\langle p, k\rangle)|$.

```
string FindBestSubsequence(StringSet $S$, $T$, int maxLength $= \infty$)
 1 string prefix, p, bestSeq;
 2 double upperBound $= \infty$, maxVal $= -\infty$, val;
 3 int $x$, $y$;
 4 PriorityQueue queue;    /* Best First Search*/
 5 push $(\varepsilon, \infty)$ to queue;
 6 while queueis not empty do
 7     let (prefix, upperBound) be the popped element from queue;
 8     if upperBound < maxVal then break;
 9     foreach $c \in \Sigma$ do
10         $p$ = prefix + $c$;    /* string concatenation */
11         $x = |S \cap L^{\text{seq}}(p)|$;
12         $y = |T \cap L^{\text{seq}}(p)|$;
13         val $= f(x, y)$;
14         if val > maxVal then
15             maxVal = val;
16             bestSeq = $p$;
17         upperBound $= max\{f(x,y), f(x,0), f(0,y), f(0,0)\}$;
18         if $|p| <$ maxLength then
19             push ($p$, upperBound) to queue;
20 return bestSeq;
```

Figure 13.2: Algorithm *FindBestSubsequence*.

We stress that the value of k is *not* given beforehand. This implies that the search space for this problem is $\Sigma^* \times \mathcal{N}$, while that of finding the best subsequence pattern is Σ^*. We remark that the above problem is NP-hard as well.

Our pruning heuristics for the purpose of restricting the number of candidate patterns is based on the following lemma that derives from Lemma 45 and Lemma 49.

Lemma 50 *For any $\langle p, k\rangle$ and $\langle u, j\rangle$ with $p, u \in \Sigma^*$ and $k, j \in \mathcal{N}$, if $p \preceq_{\text{seq}} u$ and $k \leq j$, $f(x_{\langle u,j\rangle}, y_{\langle u,j\rangle}) \leq \max\{f(x_{\langle p,k\rangle}, y_{\langle p,k\rangle}), f(x_{\langle p,k\rangle}, 0), f(0, y_{\langle p,k\rangle}), f(0, 0)\}$.*

Now we turn our attention to speeding up the matching phase of the algorithm.

Definition 44 (Computing the best window size according to f)
Input: Two sets $S, T \subseteq \Sigma^*$ of strings and a pattern string $p \in \Sigma^*$.
Output: An integer $k \in \mathcal{N}$ that maximizes the score of $f(x_{\langle p,k\rangle}, y_{\langle p,k\rangle})$, where $x_{\langle p,k\rangle} = |S \cap L^{\text{eps}}(\langle p, k\rangle)|$ and $y_{\langle p,k\rangle} = |T \cap L^{\text{eps}}(\langle p, k\rangle)|$.

This is a sub-problem of the one in Definition 43, where a pattern string p is given beforehand.

Let ℓ be the length of the longest string in $S \cup T$. A short consideration reveals that, as candidates for k, we only have to consider the values from $|p|$ up to ℓ, which results in a rather straightforward solution. In addition to that, we will give a more efficient computation method.

For strings $p, u \in \Sigma^*$, we define the *threshold value* θ of p for u by

$$\theta_{u,p} = \min\{k \in \mathcal{N} \mid u \in L^{\text{eps}}(\langle p, k\rangle)\}.$$

If there is no such value, let $\theta_{u,p} = \infty$. Note that $u \notin L^{\text{eps}}(\langle p, k\rangle)$ for any $k < \theta_{u,p}$ and $u \in L^{\text{eps}}(\langle p, k\rangle)$ for any $k \geq \theta_{u,p}$.

Definition 45 (Computing the minimum window size)
Input: Two strings $p, u \in \Sigma^*$.
Output: The threshold value $\theta_{u,p}$.

It is easy to see that the above sub-problem can be efficiently solved by using $EDASG(u)$ (see Section 10.2), in $O(|p| \cdot |u|)$ time.

The set of threshold values of $p \in \Sigma^*$ with respect to $S \subseteq \Sigma^*$ is defined as $\Theta_{S,p} = \{\theta_{u,p} \mid u \in S\}$. A key observation is that the best window size for given $S, T \subseteq \Sigma^*$ and pattern $p \in \Sigma^*$ can be found in set $\Theta_{S,p} \cup \Theta_{T,p}$ without loss of generality. Thus we can restrict the search space for the best window size to $\Theta_{S,p} \cup \Theta_{T,p}$.

From now on, we consider the numerical sequence $\{x_{\langle p,k\rangle}\}_{k=0}^{\infty}$. (We will treat $\{y_{\langle p,k\rangle}\}_{k=0}^{\infty}$ in the same way.) It clearly follows from Lemma 50 that the sequence is non-decreasing. Remark that $0 \leq x_{\langle p,k\rangle} \leq |S|$ for any k. Moreover, $x_{\langle p,l\rangle} = x_{\langle p,l+1\rangle} = x_{\langle p,l+2\rangle} = \cdots$, where l is the length of the longest string in S. Hence, we can represent $\{x_{\langle p,k\rangle}\}_{k=0}^{\infty}$ with a list having at most $\min\{|S|, l\}$ elements. We call this list a *compact representation of the sequence* $\{x_{\langle p,k\rangle}\}_{k=0}^{\infty}$ (*CRS*, for short).

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

```
string FindBestEpisode(StringSet $S$, $T$, int $\ell$)
1  string prefix, p;
2  episodePattern bestEpisode; /* 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  push $(\varepsilon, \infty)$ to queue;
8  while queueis not empty do
9      let (prefix, upperBound) be the popped element from queue;
10     if upperBound < maxVal then break;
11     foreach $c \in \Sigma$ do
12         $p = prefix + c$;    /* string concatenation */
13         $\bar{x} = S.crs(p)$;
14         $\bar{y} = T.crs(p)$;
15         $k' = \mathrm{argmax}_k\{f(x_{\langle p,k\rangle}, y_{\langle p,k\rangle})\}$ and $val = f(x_{\langle p,k'\rangle}, y_{\langle p,k'\rangle})$;
16         if val > maxVal then
17             maxVal = val;
18             bestEpisode = $\langle p, k'\rangle$;
19         $upperBound = max\{f(x_{\langle p,\infty\rangle}, y_{\langle p,\infty\rangle}), f(x_{\langle p,\infty\rangle}, 0), f(0, y_{\langle p,\infty\rangle}), f(0,0)\}$;
20         if $|p| <$ maxLength;
21         if upperBound > maxVal and $|p| < \ell$ then
22             push (p, upperBound) to queue;
23 return bestEpisode;
```

Figure 13.3: Algorithm *FindBestEpisode*.

After constructing CRSs $\bar{x}$ of $\{x_{\langle p,k\rangle}\}_{k=0}^{\infty}$ and $\bar{y}$ of $\{y_{\langle p,k\rangle}\}_{k=0}^{\infty}$, 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 finding the best threshold value.

Lemma 51 *Given $S, T \subseteq \Sigma^*$ and $p \in \Sigma^*$, we can find the best threshold value in $O((\|S\| + \|T\|) \cdot |p|)$ time.*

The whole algorithm to find the best episode pattern from two given sets of strings according to function f is shown in Figure 13.3. We add a method *crs*(p) to the data structure **StringSet** that returns CRS of $\{x_{\langle p,k\rangle}\}_{k=0}^{\infty}$, in the way mentioned above.

By Lemma 50, we can use the value $upperBound = max\{f(x_{\langle p,\infty\rangle}, y_{\langle p,\infty\rangle}), f(x_{\langle p,\infty\rangle}, 0), f(0, y_{\langle p,\infty\rangle}), f(0,0)\}$ to prune branches in the search tree computed at line 19. Notice that the value $max\{f(x_{\langle p,k\rangle}, y_{\langle p,k\rangle}), f(x_{\langle p,k\rangle}, 0), f(0, y_{\langle p,k\rangle}), f(0,0)\}$ is *not* appropriate as *upperBound*. Note also that $x_{\langle p,\infty\rangle}$ and $y_{\langle p,\infty\rangle}$ can be extracted from $\bar{x}$ and $\bar{y}$ in constant time, respectively.

13.5 Finding Best VLDC Patterns

Let $q, u \in \Pi$, where $\Pi = (\Sigma \cup \{\star\})^*$. We write as $q \preceq_{\text{vldc}} u$ if u can be obtained by replacing $\star$'s in q with some elements in Π.

Definition 46 Let $q \in \Pi$. The *VLDC language* of q is defined by

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

According to the above definition, we have the following lemma.

Lemma 52 *For any $q, u \in \Pi$, if $q \preceq_{\text{vldc}} u$, then $L^{\text{vldc}}(q) \supseteq L^{\text{vldc}}(u)$.*

Definition 47 (Finding the best VLDC pattern according to f)
Input: Two sets $S, T \subseteq \Sigma^*$ of strings.
Output: A VLDC pattern $q \in \Pi$ that maximizes the score of $f(x_q, y_q)$, where $x_q = |S \cap L^{\text{vldc}}(q)|$ and $y_q = |T \cap L^{\text{vldc}}(q)|$.

The problem is known to be NP-hard [56], and thus we essentially have exponentially many candidates. Therefor, we reduce the number of candidates by using the pruning heuristics as well.

By Lemma 45 and Lemma 52, we have the next lemma, basing on which we can perform the pruning heuristic to speed up our algorithm.

Lemma 53 *For any $q, u \in \Pi$, if $q \preceq_{\text{vldc}} u$, then $f(x_u, y_u) \leq \max\{f(x_q, y_q), f(x_q, 0), f(0, y_q), f(0, 0)\}$.*

The general concept of the algorithm to solve the problem of Definition 47 is the same as the one in Figure 13.2.

To speed up the matching phase of the algorithm, we consider the following problem.

Definition 48 (Counting the matched VLDC patterns)
Input: A set $S \subseteq \Sigma^*$ of strings.
Query: A pattern $q \in \Pi$.
Output: The cardinality of set $S \cap L^{\text{vldc}}(q)$.

This is a sub-problem of the one in Definition 47. The first idea is to use a DFA accepting $L^{\text{vldc}}(q)$ and run it over each string in S, as mentioned in Section 10.3. Then, $|S \cap L^{\text{vldc}}(q)|$ can be computed in $O(|q|)$ preprocessing time and in $O(\|S\|)$ running time.

Another idea is to construct $WDAWG(s)$ for each string $s \in S$. This way, $|S \cap L^{\text{vldc}}(q)|$ can be computed in $O(N)$ preprocessing time and $O(|S| \cdot |q|)$ running time, where $N = \sum_{s \in S} |s|^2$.

13.6 Finding Best VLDC Patterns in Window

Recall the VLDC pattern matching problem with window size, introduced in Section 10.4.

Definition 49 For a pair $\langle q, k\rangle$ with $q \in \Pi$ and $k \in \mathcal{N}$, its language is defined by

$$L^{\text{vldcw}}(\langle q, k\rangle) = \{w \in \Sigma^* \mid \langle q, k\rangle \text{ matches } w\}.$$

According to the above definition, we have the following lemma.

Lemma 54 *For any $\langle q, k\rangle$ and $\langle u, j\rangle$ with $q, u \in \Pi$ and $k, j \in \mathcal{N}$, if $q \preceq_{\text{vldc}} u$ and $k \leq j$, then $L^{\text{vldcw}}(\langle q, k\rangle) \supseteq L^{\text{vldcw}}(\langle u, j\rangle)$.*

The problem to be tackled is formalized as follows.

Definition 50 (Finding the best VLDC pattern and window size according to f)

Input: Two sets $S, T \subseteq \Sigma^*$ of strings.
Output: A pair $\langle q, k\rangle$ with $q \in \Pi$ and $k \in \mathcal{N}$ that maximizes the score of $f(x_{\langle q,k\rangle}, y_{\langle q,k\rangle})$, where $x_{\langle q,k\rangle} = |S \cap L^{\text{vldcw}}(\langle q, k\rangle)|$ and $y_{\langle q,k\rangle} = |T \cap L^{\text{vldcw}}(\langle q, k\rangle)|$.

We stress that the value of k is *not* given beforehand, i.e., we compute not only q but also k with which the score of function f is maximum. Therefore, the search space of this problem is $\Pi \times \mathcal{N}$, while that of the problem in Definition 47 is Π. We remark that this problem is also NP-hard.

By Lemma 45 and Lemma 54, we achieve the following lemma that plays a key role in the heuristics for pruning the search tree.

Lemma 55 *For any $\langle q, k\rangle$ and $\langle u, j\rangle$ with $q, u \in \Pi$ and $k, j \in \mathcal{N}$, if $q \preceq_{\text{vldc}} u$ and $k \leq j$, $f(x_{\langle u,j\rangle}, y_{\langle u,j\rangle}) \leq \max\{f(x_{\langle q,k\rangle}, y_{\langle q,k\rangle}), f(x_{\langle q,k\rangle}, 0), f(0, y_{\langle q,k\rangle}), f(0, 0)\}$.*

The general concept of the algorithm for finding the best pair of $q \in \Pi$ and $k \in \mathcal{N}$ is the same as the one in Figure 13.3.

Speed-up of the matching phase of the algorithm can be achieved by quickly answering the following sub-problem of Definition 50.

Definition 51 (Computing the best window size for VLDC pattern according to f)

Input: Two sets $S, T \subseteq \Sigma^*$ of strings and a VLDC pattern $q \in \Pi$.
Output: An integer $k \in \mathcal{N}$ that maximizes the score of $f(x_{\langle q,k\rangle}, y_{\langle q,k\rangle})$, where $x_{\langle q,k\rangle} = |S \cap L^{\text{vldcw}}(\langle q, k\rangle)|$ and $y_{\langle q,k\rangle} = |T \cap L^{\text{vldcw}}(\langle q, k\rangle)|$.

For a string $u \in \Sigma^*$ and VLDC pattern $q \in \Pi$, we define the *threshold value* θ' of q for u by

$$\theta_{u,q} = \min\{k \in \mathcal{N} \mid u \in L^{\text{vldcw}}(\langle q, k\rangle)\}.$$

If there is no such value, let $\theta'_{u,q} = \infty$. Note that $u \notin L^{\text{vldcw}}(\langle q, k\rangle)$ for any $k < \theta'_{u,q}$ and $u \in L^{\text{vldcw}}(\langle q, k\rangle)$ for any $k \geq \theta'_{u,q}$.

The set of threshold values for $q \in \Pi$ with respect to $S \subseteq \Sigma^*$ is defined as $\Theta'_{S,q} = \{\theta'_{u,q} \mid u \in S\}$. A key observation is that the best window size for given sets $S, T \subseteq \Sigma^*$ of strings and a VLDC pattern $q \in \Pi$ can be found in set $\Theta'_{S,q} \cup \Theta'_{T,q}$ without loss of generality. Thus we can restrict the search space for the best window size to $\Theta'_{S,q} \cup \Theta'_{T,q}$. It is therefore important to quickly solve the following sub-problem.

Definition 52 (Computing the minimum window size for VLDC pattern)
Input: A string $u \in \Sigma^*$ and VLDC pattern $q \in \Pi$.
Output: The threshold value $\theta'_{u,q}$.

This is a sub-problem of Definition 51. We remark that this problem can be solved in $O(|u| \cdot |q|)$ running time by any method introduced in Section 10.4.

13.7 Computational Experiments

The algorithms were implemented in the Objective Caml Language. All calculations were performed on a Desktop PC with dual Xeon 2.2GHz CPU (though our algorithms only utilize single CPU) with 1GB of main memory running Debian Linux. In all the experiments, the entropy information gain is used as the score for which the search is conducted.

13.7.1 Artificial Data

We first tested our algorithms on an artificial dataset. The datasets were created as follows: The alphabet was set to $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}\}$. We then randomly generate strings over

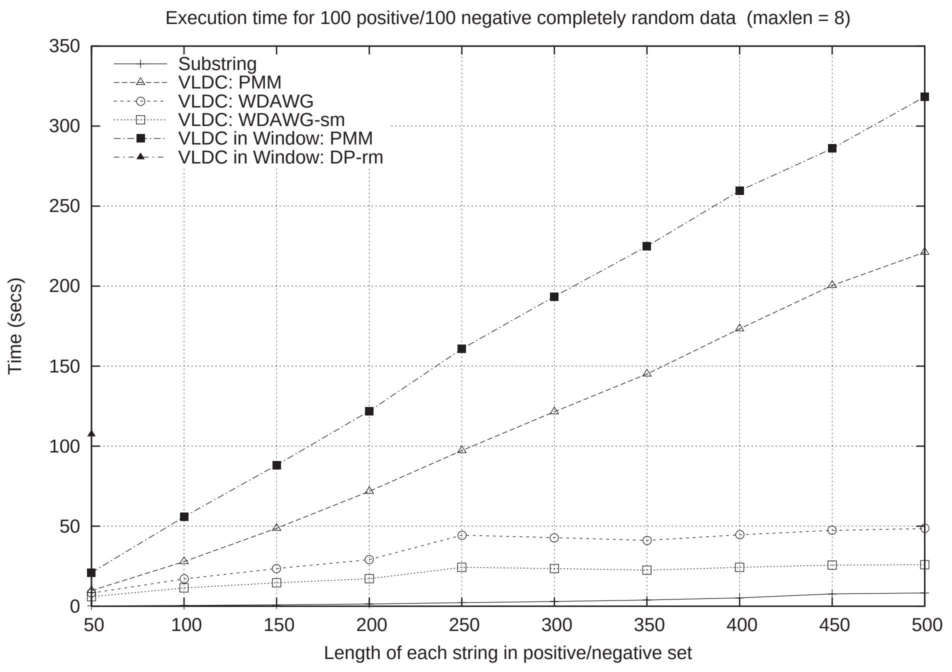


Figure 13.4: Execution time (in seconds) for artificial data for different lengths of the examples. The maximum length of patterns to be searched for is set to 8. WDAWG-sm is matching using the WDAWG with state memoization. DP-rm is matching using the dynamic programming table with row memoization. Only one point is shown for DP-rm here, since a greater size caused memory swapping, and the computation was not likely to end in a reasonable amount of time.

Σ of length l. We created 3 types of datasets: 1) a completely random set, 2) a set where a randomly chosen VLDC pattern $\star$`ccd`$\star$`a`$\star$`ddad`$\star$ is embedded in the positive examples, and 3) a set where a pair of a VLDC pattern and a window size $\langle\star\mathtt{ccd}\star\mathtt{a}\star\mathtt{ddad}\star, 19\rangle$ is embedded in the positive examples. In 2) and 3), a randomly generated string is used as a positive example if the pattern matches it, and used as a negative example otherwise, until both positive and negative set sizes are n. Examples for which the set size exceeds n are discarded.

Figure 13.4 and Figure 13.5 show the execution times for the completely random dataset for different l and n, respectively. We can see that the execution time grows linearly in n and l as expected, although the effect of pruning seems to take over for VLDC patterns in the left graph, when the length of each sequence is long. Searching for VLDC patterns and window sizes using dynamic programming with row memoization, does not perform very well.

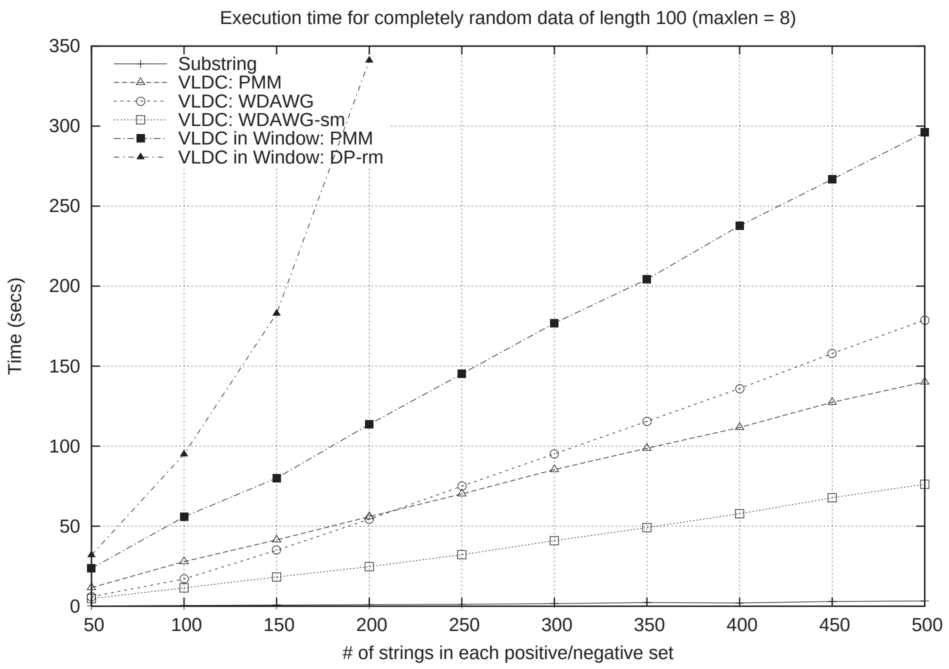


Figure 13.5: Execution time (in seconds) for artificial data for different number of examples in each positive/negative set. The maximum length of patterns to be searched for is set to 8. WDAWG-sm is matching using the WDAWG with state memoization. DP-rm is matching using the dynamic programming table with row memoization.

Figure 13.6, Figure 13.7, and Figure 13.8 show the execution times for different maximum lengths of VLDC patterns to look for, for the 3 datasets, respectively (The length of a VLDC pattern is defined as the length of the pattern representation, excluding any $\star$'s on the ends). We can see that the execution time grows exponentially as we increase the maximum pattern length searched for, until the pruning takes effect. Figure 13.9 shows the effect of performance of an exhaustive search, run on the completely random dataset, compared to searches with the branch and bound pruning for the different datasets. The pruning is more effective when it is more likely to have a good solution.

13.7.2 Real Data

To show the usefulness of VLDC patterns and windows, we also tested our algorithms on actual protein sequences. We use the data available at `http://www.cbs.dtu.dk/services/TargetP/`, which consists of protein sequences which are known to contain *protein sorting signals*, that is, (in many cases) a short amino acid sequence segment which holds the information which enables the protein to be carried to specified compartments inside the cell. The dataset for plant proteins consisted of: 269 sequences with signal peptide (SP), 368 sequences with mitocondrial targeting peptide (mTP), 141 sequences with chloroplast transit peptide (cTP), and 162 "Other" sequences. The average length of the sequences was around 419, and the alphabet is the set of 20 amino acids.

Using the signal peptides as positive examples, and all others as negative examples, we searched for the best pair $\langle p, k\rangle$ with maximum length of 10 using PMMs. To limit the alphabet size, we classify the amino acids into 3 classes $\{0, 1, 2\}$, according to the hydropathy index [48]. The most hydrophobic amino acids {A, M, C, F, L, V, I} (hydropathy ≥ 0.0) are converted to 0, {P,Y,W,S,T,G} ($-3.0 \leq$ hydropathy < 0.0) to 1, and {R, K, D, E, N, Q, H} (hydropathy < -3.0) to 2. We obtained the pair $\langle 0{\star}00{\star}00000{\star}, 26\rangle$, which occurs in $213/269 = 79.2\%$ of the sequences with SP, and $26/671 = 3.9\%$ of the other sequences. The calculation took exactly 50 minutes. This pattern can be interpreted as capturing the well known hydrophobic h-region of SP [75]. Also, the VLDC pattern suggests that the match occurs in the first 26 amino acid residues of the protein, which is natural since SP, mTP, cTP are known to be *N-terminal sorting signals*, that is, they are known to appear near the head of the protein sequence. A best substring search quickly finds the pattern ${\star}00000001{\star}$ in 36 seconds, but only gives us a classifier that matches $152/269 = 56.51\%$ of the SP sequences, and $41/671 = 6.11\%$ of the others.

For another example, we use the mTP set as positive examples, and the SP and Other sets as negative examples. This time, we convert the alphabet according to the net charge of the amino acid. Amino acids {D, E} (negative charge) are converted to 0, {K, R} (positive charge) to 1, and the rest {A, L, N, M, F, C, P, Q, S, T, G, W, H, Y, I, V} to 2. The calculation took about 21 minutes and we obtain the pair $\langle 2{\star}1{\star}1{\star}2221{\star}, 28\rangle$ which occurs in $334/368 = 90.76\%$ of the mTP sequences and ($73/431 = 16.94\%$) of the SP and Other sequences. This pattern can also be regarded as capturing existing knowledge about mTPs [76]: They are fairly abundant in K or R, but do not contain much D or

E. The pattern also suggests a periodic appearance of K or R, which is a characteristic of an amphiphilic α-helix that mTPs are reported to have. A best substring search finds pattern $\star 212221 \star$ in 20 seconds, which gives us a classifier that matches $318/368 = 86.41\%$ of sequences with mTP and $255/431 = 59.16\%$ of the other sequences.

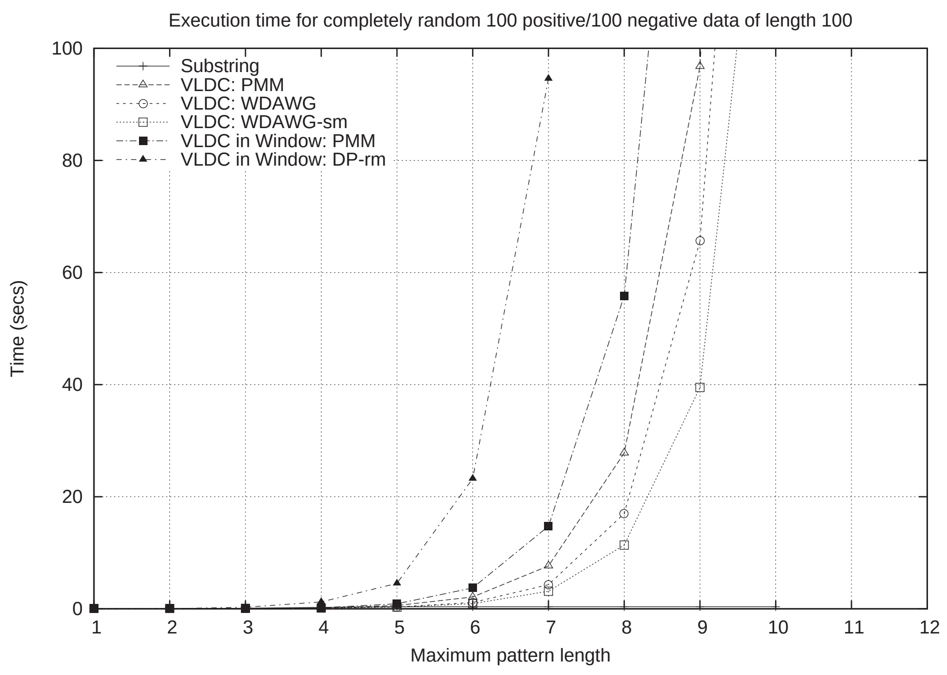


Figure 13.6: Execution time (in seconds) for artificial data for different maximum lengths of patterns to be searched for with completely random data.

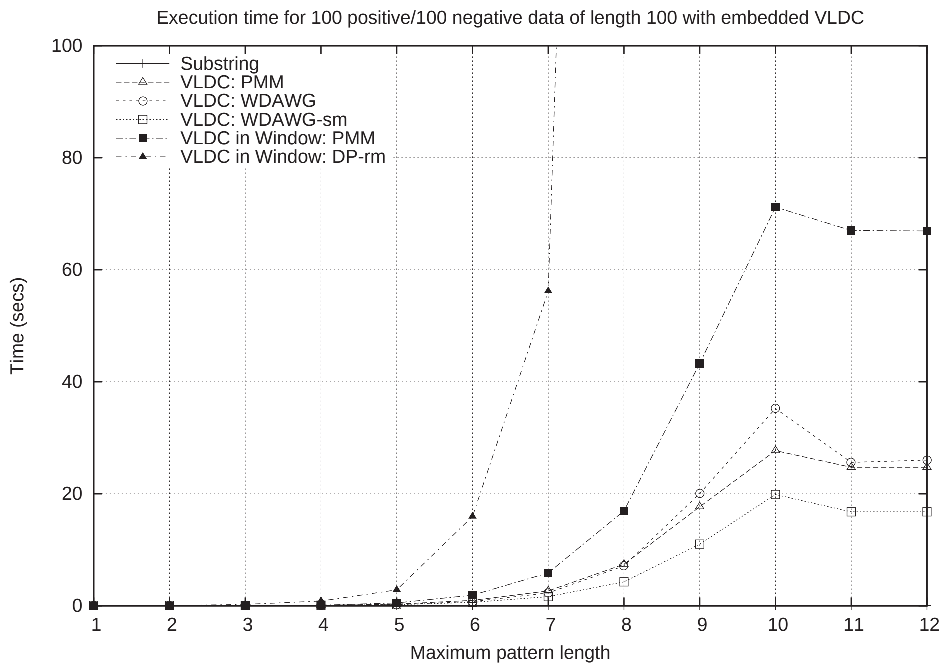


Figure 13.7: Execution time (in seconds) for artificial data for different maximum lengths of patterns to be searched for with VLDC embedded data.

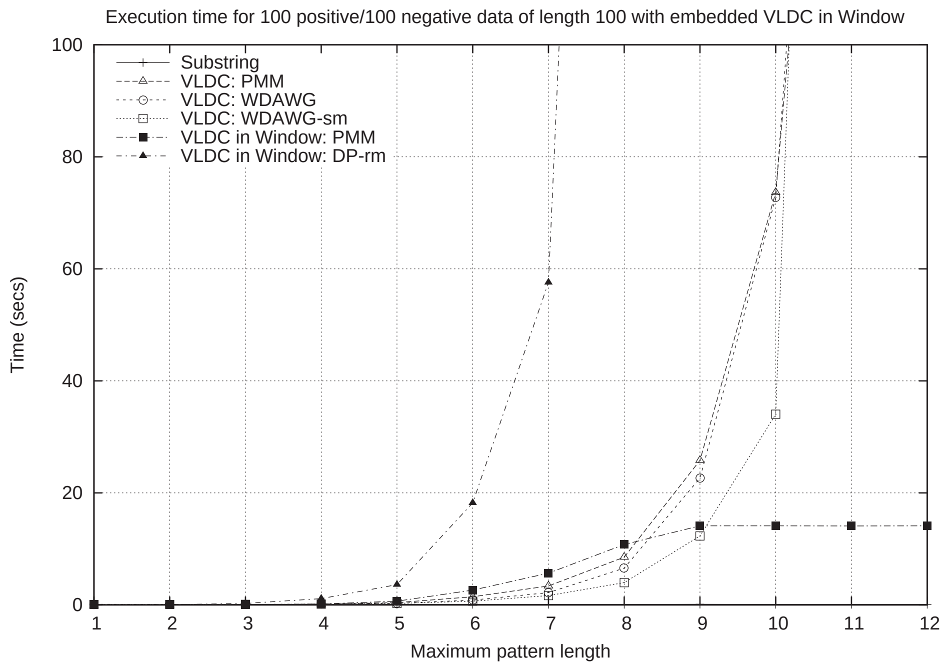


Figure 13.8: Execution time (in seconds) for artificial data for different maximum lengths of patterns to be searched for with VLDC and window size embedded data.

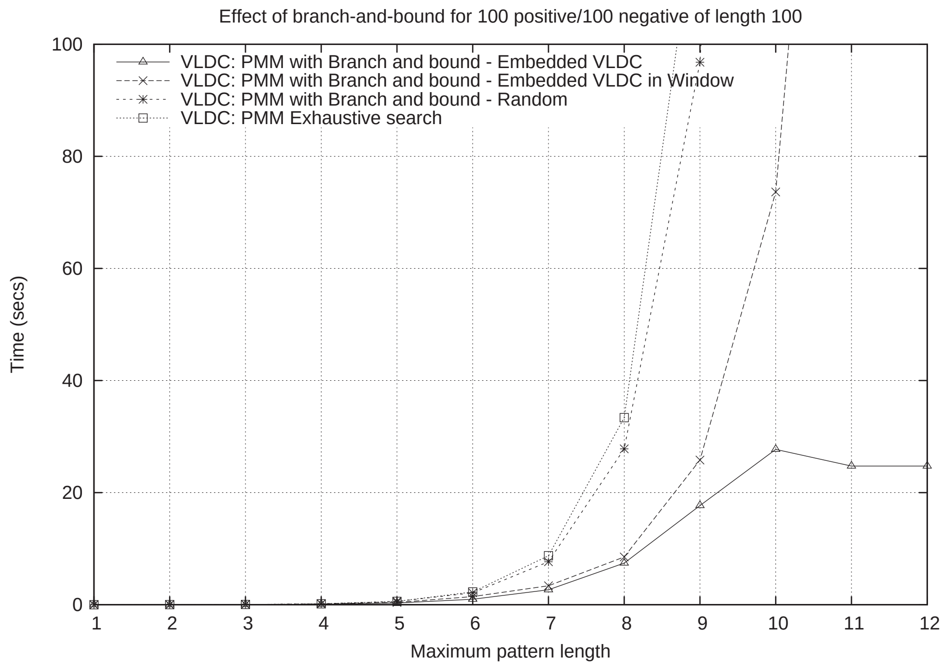


Figure 13.9: The effect of pruning of the search space for the different data sets, compared to exhaustive search on the completely random dataset.

Chapter 14

Concluding Remarks and Future Perspectives

In this thesis we studied various algorithms on string processing. First we presented several algorithms for constructing index structures that contribute efficient solutions of the substring pattern matching problem. The problem is the most fundamental as well as the most important. Index structures, which effectively enable us to solve the problem in $O(|p|)$ running time for any given pattern string p, are therefore quite important.

We declare that the algorithm we presented in Chapter 4 has all the desired properties: it constructs *CDAWGs in linear time*, *on-line*, and is *applicable to inputs of sets of strings.* We can see the list in Table 4.1 now complete, which implies that our algorithm has filled the missing piece and even could be seen to be the final solution to constriction of index structures, also in the sense that CDAWGs require least space as index structures of these kinds. Moreover, it gave us a unified view to the algorithms for constructing index structures as described in Chapter 9.

In Chapter 5 we gave a detailed description of how to construct a CDAWG for a set of strings. We first showed that the algorithm of Chapter 4 that builds $CDAWG'(w)$ for a single string w can be easily modified so that it constructs, for any set S of strings, $CDAWG'(S)$ in $O(\|S\|)$ time. It is then improved so as to construct $CDAWG'(S)$ *in* $O(|T|)$ *time* when the set S is given in the form of trie T. We stress that $|T|$ is much less than $\|S\|$ when strings in S share many and long prefixes, and therefore in such case it runs faster than the first approach introduced in the chapter. We also emphasize that applying the generic algorithm of Chapter 9 to this scheme, we will be able to construct $STree'(S)$ and $DAWG(S)$ in $O(|T|)$ time as well.

Chapter 6 was devoted to the algorithm for *constructing and maintaining CDAWGs in the sliding window mechanism.* The algorithm could be a space-efficient alternative to Larsson's algorithm for suffix trees for a sliding window [49]. The design of our algorithm is a combination of the on-line CDAWG construction algorithm of Chapter 4 that moves ahead the rightmost position of the window, and our original techniques to move the leftmost position ahead. This algorithm is promising to contribute to reducing space requirements in PPM style text compression scheme.

It is still an open problem whether conversion of $CDAWG'(bu)$ to $CDAWG'(u)$ can be done in (amortized) constant time for any character b and string u. Also, it is surely worth considering *DAWGs for a sliding window* where labels of edges of DAWGs are single characters. This is really a big advantage in the scheme of a sliding window since we would not need credit issuing then, which is surely time-consuming and makes the algorithm rather complicated. However, we hold a strong belief that conversion of $DAWG(bu)$ into $DAWG(u)$ cannot be done in (amortized) constant time, either. Thus we will need some alternative way, like in case of CDAWGs.

We gave an *on-line algorithm to construct SCDAWGs* in Chapter 7, which runs in linear time. Since the space-requirement of SCDAWGs is strictly smaller than that of affix trees, our algorithm contributes reduction of memory space needed for construction of bidirectional index structures. The index structures mentioned in Chapter 7, including affix trees, all have certain dualities. One interesting question is whether there is a clear duality in the suffix arrays of a string and its reversal.

In Chapter 8 we presented a linear-time algorithm for *bidirectional construction of suffix trees*, which would be an alternative of Maaß's algorithm for affix trees. We also showed that our algorithm is capable of bidirectional construction of DAWGs. To tell the truth, however, the algorithm does not always update $DAWG(u)$ to $DAWG(bua)$ with $a, b \in \Sigma$ and $u \in \Sigma^*$. We are almost sure that it is impossible to do the above update in (amortized) constant time, basing on the observation in Chapter 11. Still, this is only a conjecture, as a matter of fact.

As a related work a problem of *inferring strings from graphs*, which we proposed in [40], can be raised. There we are given a dag G, and infer a string that suits the graph under some condition. Firstly, we studied the problem of finding a string w such that $DASG(w)$ is isomorphic to a given graph G. We presented a linear-time algorithm to solve the problem. Secondly, we considered DAWGs in terms of the string inference problem.

We proposed an algorithm to solve this problem in quadratic time. We also showed that the string inference problem for DAWGs can be reduced to the *substring equations problem* where we are given equations for strings that should be substrings of a string. The most interesting open problem at the moment is whether or not any faster algorithm for solving the problem for DAWGs exists, possibly in $O(n \log n)$ or $O(n)$ time. Also, *factor oracles* [2] are surely worth to consider in the context of string inference. Factor oracles may be simpler than DAWGs, but more complicated than DASGs. Therefore it is of great interest if the problem can be solved in linear time for factor oracles. Another context of string inference would be to infer stings from *arrays*. Franěk et al. [23] presented a method to check if an integer array f is a border array for some string w. They showed an on-line linear time algorithm to verify if f is a border array for some string w on an unbounded size alphabet. Duval et al. gave an on-line linear time algorithm for bounded size alphabet [20]. We are also interested in the problem of examining whether a given integer array f is the suffix array of some string w. Is it possible to solve the string inference problem for suffix arrays in linear time?

Chapter 10 was devoted to the introduction of several advanced pattern matching problems. There we proposed *WDAWGs which allow us to solve the VLDC pattern matching problem in time linear to the length of a given pattern*. What should be emphasized here is that WDAWGs are the first index structure for solving the VLDC pattern matching problem. Due to the recent high necessity of analysis of genomic sequences which often contain noises, some efficient pattern matching method allowing *errors* has been sought. VLDC pattern matching is one of the most powerful "weapons" for this, and therefore, the importance of WDAWGs is quite high. We also presented the problem of *VLDC pattern matching within a window*, which is a generalization of the episode pattern matching problem. The introduction of the window helps us avoid unwanted, too long matches of VLDC patterns.

In Chapter 11 we gave an *on-line algorithm to directly build MASDAWGs*, which are inherently the same structure as WDAWGs. We showed that the algorithm runs in time linear to the output size. We also presented a more space-economical algorithm to construct MASDAWGs in Chapter 12. The algorithm reads a given string right to left, in contrast to the above on-line algorithm that processes a given string left to right. The all-suffixes version of CDAWGs, named *MASCDAWGs*, were introduced in this chapter as well. Since the space requirement of CDAWGs is smaller than DAWGs, the new structure surely contribute to further reduction of space requirement of index structures for all suffixes of a string. We proposed an algorithm that constructs $MASCDAWG(w)$ directly, in time proportional to the output size, processing w right to left.

We studied pattern discovery from textual data in Chapter 13. We first produced *a practical algorithm to find the best episode patterns* in the scheme of the BONSAI system [63], where we are given two sets of strings and find the best pattern for the purpose of separating the two sets. The search space is much bigger than in the case of the original version of BONSAI, and thus we utilized *EDASGs* to speed up the matching phase of the algorithm and employed the pruning heuristics to restrict the number of candidate patterns. Secondly, we proposed an efficient algorithm that discovers the *best VLDC pattern* to distinguish two given sets of strings. We used WDAWGs in order to accelerate the matching of given patterns with text strings, and also gave an alternative methods with the use of pattern matching machines. Finally, we introduced an algorithm to discover the *best VLDC pattern within a window.* We presented three methods for finding the best window size using dynamic programming, pattern matching machines, and WDAWGs. We also showed the experimental results which are good enough to convince us the effect of the pruning heuristics and the usefulness of VLDC patterns within windows. As a future work, we are planning to apply the scheme of the *edit distance* to the algorithm. The pattern will be a tuple $\langle (p, k), d \rangle$ where p is a VLDC pattern, k is a window size, and d is a *hamming distance.* Namely, we will look for the VLDC pattern within a window which, within k hamming distance, matches as many strings as possible in one set and as few strings as possible in the other.

Bibliography

[1] M. I. Abouelhoda, E. Ohlebusch, and S. Kurtz. Optimal exact string matching based on suffix arrays. In *Proc. 9th International Symposium on String Processing and Information Retrieval (SPIRE'02)*, volume 2476 of *Lecture Notes in Computer Science*, pages 31–43. Springer-Verlag, 2002.

[2] C. Allauzen, M. Crochemore, and M. Raffinot. Factor oracle: A new structure for pattern matching. In *Proc. 26th Annual Conference on Current Trends in Theory and Practice of Informatics (SOFSEM'99)*, volume 1725 of *Lecture Notes in Computer Science*, pages 291–306. Springer-Verlag, 1999.

[3] D. Angluin. Finding patterns common to a set of strings. *Journal of Computer System Sciences*, 21:46–62, 1980.

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

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

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

[7] H. Bannai, K. Iida, A. Shinohara, M. Takeda, and S. Miyano. More speed and more pattern variations for knowledge discovery system BONSAI. In *Proc. 12th International Conference on Genome Informatics (GIW'01)*, pages 454–455. Universal Academy Press, 2001.

[8] H. Bannai, S. Inenaga, A. Shinohara, M. Takeda, and S. Miyano. A string pattern regression algorithm and its application to pattern discovery in long introns. In

Proc. 13th International Conference on Genome Informatics (GIW'02), pages 3–11. Universal Academy Press, 2002.

[9] 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.

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

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

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

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

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

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

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

[17] M. Crochemore and Z. Troníček. On the size of dasg for multiple texts. In *Proc. 9th International Symposium on String Processing and Information Retrieval (SPIRE'02)*, volume 2476 of *Lecture Notes in Computer Science*, pages 58–64. Springer-Verlag, 2002.

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

[19] G. Das, R. Fleischer, L. Gasieniec, D. Gunopulos, and J. Kärkkäinen. Episode matching. In *Proc. 8th Annual Symposium on Combinatorial Pattern Matching (CPM'97)*, volume 1264 of *Lecture Notes in Computer Science*, pages 12–27. Springer-Verlag, 1997.

[20] J.-P. Duval, T. Lecroq, and A. Lefevre. Border array on bounded alphabet. In *Proc. The Prague Stringology Conference '02 (PSC'02)*, pages 28–35. Czech Technical University, 2002.

[21] M. Farach. Optimal suffix tree construction with large alphabets. In *Proc. 38th Annual Symposium on Foundations of Computer Science (FOCS'97)*, pages 137–143. IEEE Computer Society, 1997.

[22] E. R. Fiala and D. H. Greene. Data compression with finite windows. *Communications of the ACM*, 32(4):490–505, 1989.

[23] F. Franěk, S. Gao, W. Lu, P. J. Ryan, W. F. Smyth, Y. Sun, and L. Yang. Verifying a border array in linear time. *J. Comb. Math. Comb. Comput.*, pages 223–236, 2002.

[24] 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.

[25] 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.

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

[27] 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 *Progress in Discovery Science*, volume 2281 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 2002.

[28] M. Hirao, H. Hoshino, A. Shinohara, M. Takeda, and S. Arikawa. A practical algorithm to find the best subsequence patterns. In *Proc. The Third International Conference on Discovery Science (DS'00)*, volume 1967 of *Lecture Notes in Artificial Intelligence*, pages 141–154. Springer-Verlag, 2000.

[29] M. Hirao, S. Inenaga, A. Shinohara, M. Takeda, and S. Arikawa. A practical algorithm to find the best episode patterns. In *Proc. The Fourth International Conference on Discovery Science (DS'01)*, volume 2226 of *Lecture Notes in Artificial Intelligence*, pages 435–440. Springer-Verlag, 2001.

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

[31] H. Hoshino, A. Shinohara, M. Takeda, and S. Arikawa. Online construction of subsequence automata for multiple texts. In *Proc. 7th International Symposium on String Processing and Information Retrieval (SPIRE'00)*, pages 146–152. IEEE Computer Society, 2000.

[32] L. Hui. Color set size problem with applications to string matching. In *Proc. 3rd Annual Symposium on Combinatorial Pattern Matching (CPM'92)*, volume 644 of *Lecture Notes in Computer Science*, pages 230–243. Springer-Verlag, 1992.

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

[34] S. Inenaga. Bidirectional construction of suffix trees. In *Proc. The Prague Stringology Conference '02 (PSC'02)*, pages 75–87. Czech Technical University, 2002.

[35] S. Inenaga, H. Bannai, A. Shinohara, M. Takeda, and S. Arikawa. Discovering best variable-length-don't-care patterns. In *Proc. The Fifth International Conference on Discovery Science (DS'02)*, volume 2534 of *Lecture Notes in Computer Science*, pages 86–97. Springer-Verlag, 2002.

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

[37] 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.

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

[39] S. Inenaga, H. Hoshino, A. Shinohara, M. Takeda, S. Arikawa, G. Mauri, and G. Pavesi. On-line construction of compact directed acyclic word graphs. In *Proc. 12th Annual Symposium on Combinatorial Pattern Matching (CPM'01)*, volume 2089 of *Lecture Notes in Computer Science*, pages 169–180. Springer-Verlag, 2001.

[40] S. Inenaga, A. Shinohara, and M. Takeda. Inferring strings from graphs. Technical Report DOI-TR-CS-215, Department of Informatics, Kyushu University, 2003.

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

[42] S. Inenaga, A. Shinohara, M. Takeda, H. Bannai, and S. Arikawa. Space-economical construction of index structures for all suffixes of a string. In *Proc. 27th International Symposium on Mathematical Foundations of Computer Science (MFCS'02)*, volume 2420 of *Lecture Notes in Computer Science*, pages 341–352. Springer-Verlag, 2002.

[43] S. Inenaga, M. Takeda, A. Shinohara, H. Hoshino, and S. Arikawa. The minimum DAWG for all suffixes of a string and its applications. In *Proc. 13th Annual Symposium on Combinatorial Pattern Matching (CPM'02)*, volume 2373 of *Lecture Notes in Computer Science*, pages 153–167. Springer-Verlag, 2002.

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

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

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

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

[48] J. Kyte and R. Doolittle. A simple method for displaying the hydropathic character of a protein. *Journal of Molecular Biology*, 157:105–132, 1982.

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

[50] N. J. Larsson. *Structures of String Matching and Data Compression.* PhD thesis, Lund University, 1999.

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

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

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

[54] H. Mannila, H. Toivonen, and A. I. Verkamo. Discovering frequent episode in sequences. In *Proc. 1st International Conference on Knowledge Discovery and Data Mining*, pages 210–215. AAAI Press, 1995.

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

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

[57] A. Moffat. Implementing the PPM data compression scheme. *IEEE Trans. Commun.*, 38(11):1917–1921, 1990.

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

[59] J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1:81–106, 1986.

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

[61] 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.

[62] T. Shibuya. Constructing the suffix tree of a tree with a large alphabet. In *Proc. 10th Annual International Symposium on Algorithms and Computation (ISAAC'99)*, volume 1741 of *Lecture Notes in Computer Science*, pages 225–236. Springer-Verlag, 1999.

[63] S. Shimozono, A. Shinohara, T. Shinohara, S. Miyano, S. Kuhara, and S. Arikawa. Knowledge acquisition from amino acid sequences by machine learning system BONSAI. *Transactions of Information Processing Society of Japan*, 35(10):2009–2018, 1994.

[64] A. Shinohara, M. Takeda, S. Arikawa, M. Hirao, H. Hoshino, and S. Inenaga. Finding best patterns practically. In *Progress in Discovery Science*, volume 2281 of *Lecture Notes in Artificial Intelligence*, pages 307–317. Springer-Verlag, 2002.

[65] T. Shinohara. Polynomial-time inference of pattern languages and its applications. In *Proc. 7th IBM Symposium on Mathematical Foundations of Computer Science (MFCS'82)*, pages 191–209, 1982.

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

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

[68] 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 Computer Science*, 2002. (to appear).

[69] Z. Troníček. Problems related to subsequences and supersequences. In *Proc. 6th International Symposium on String Processing and Information Retrieval (SPIRE'99)*, pages 199–205. IEEE Computer Society, 1999.

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

[71] Z. Troníček and B. Melichar. Directed acyclic subsequence graph. In *Proc. The Prague Stringology CLub Workshop '98(PSCW'98)*, pages 107–118. Czech Technical University, 1998.

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

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

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

[75] G. von Heijne. The signal peptide. *Journal of Membrane Biology*, 115:195–201, 1990.

[76] G. von Heijne, J. Steppuhn, and R. G. Herrmann. Domain structure of mitochondrial and chloroplast targeting peptides. *European Journal of Biochemistry*, 180:535–545, 1989.

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