On Compact Directed Acyclic Word Graphs

Maxime Crochemore and Renaud Vérin

Institut Gaspard Monge
Université de Marne-La-Vallée,
2, rue de la Butte Verte, F-93160 Noisy-Le-Grand.
http://www-igm.univ-mlv.fr

Abstract. The Directed Acyclic Word Graph (DAWG) is a space-efficient data structure to treat and analyze repetitions in a text, especially in DNA genomic sequences. Here, we consider the Compact Directed Acyclic Word Graph of a word. We give the first direct algorithm to construct it. It runs in time linear in the length of the string on a fixed alphabet. Our implementation requires half the memory space used by DAWGs.

1 Introduction

One of the most surprising facts related to pattern matching and discovered by Ehrenfeucht *et al.* [2] is that the size of the minimal automaton accepting the suffixes of a word is linear. The surprise is due to the maximal number of subwords that may occur in a word: it is quadratic according to the length of the word. This is obviously true if the alphabet is unbounded, but still holds if the alphabet contains at least two letters. In addition to the previous result, Ehrenfeucht *et al.* proved that the automaton can be built in linear time, which is indeed a consequence of the previous fact but does not come readily from it.

In the present article, we consider the compact implementation of the automaton and show that it has a direct construction that runs in linear time. Fast and space-economical methods for this construction are important because the automaton serves as an index on the underlying word, and, as such, is involved in several combinatorial algorithms on words.

Historically, the first linear-size graph to represent the subwords of a word, called the *Directed Acyclic Word Graph* (DAWG), was described in [2] together with a linear-time construction. When terminal states are added to the DAWG, as shown in [8], the structure becomes the minimal automaton accepting the suffixes of the word. Regarded as an automaton accepting the subwords of the word, *i.e.* setting all states as terminal states, the DAWG is not always a minimal automaton. Indeed, this latter automaton can be slightly smaller, but its construction satisfies the same properties ([8, 3, 9]) though the algorithms become a bit more tricky.

Basically, DAWGs provide an implementation of indexes on texts [4]. The index on a text T helps searching it for various patterns. For instance, it leads to an efficient solution to the string-matching problem, searching text T for a word w. The typical running time of a query is $\mathcal{O}(|w|)$ on a fixed alphabet, and is $\mathcal{O}(|w| \log |\Sigma|)$ if the alphabet Σ of the text is unbounded.

Many other efficient solutions to problems on words are applications of DAWGs. They include (see [12]): computing the number of subwords of a word, computing the longest repeated subword of a word, backward DAWG-matching, finding repetitions in words [6], searching for a square [7, 9], computing the longest common subword of a finite set of words and on-line subword matching [10], approximate string-matching [21].

The *suffix tree* is an alternative representation of the subwords of a word that shares with the DAWG essentially the same applications. McCreight [18] introduced the notion and gave an efficient construction after the seminal work of Weiner [22] on a similar structure.

Suffix trees have been more extensively studied than DAWGs, probably because they display positions of the word in a simpler way although the branching from nodes is not uniform as it is from states of DAWGs. Apostolico [1] lists over forty references on suffix trees, and Manber and Myers [17] mention several others (see also [19]). Several variants or implementations of suffix trees have been developed, like *suffix arrays* [17], *PESTry* [16], *suffix cactus* [15], or *suffix binary search trees* [14]. Ukkonen [20] designs an on-line construction of suffix trees, and Farach [13] proposes a novel approach leading to a linear-time construction on integer alphabets.

In computational biology, DNA sequences are often only viewed as words over the alphabet $\{a, c, g, t\}$ of nucleotides. In this form, they are objects for linguistic and statistic analysis. For this purpose, suffix automata (or suffix trees) are extremely useful data structures, but the bottleneck to using them is their size. The indexes has to be kept in main memory and their sizes limit their use. The size of available sequences is steadily growing, and therefore saving memory space is wanted both for the construction of the index and for its use.

The *Compact Directed Acyclic Word Graph* (CDAWG) keeps the direct access to information while requiring less memory space. The structure has been introduced by Blumer *et al.* [4, 5]. The implementation is obtained by deleting all states of outdegree one and their corresponding transitions (excepting terminal states).

We present an algorithm that builds directly compact DAWGs. This construction avoids constructing the DAWG first, which makes it suitable for the presently available DNA sequences (about 1.5 million nucleotides long for the longest sequences). Experiments show that our implementation saves half of the memory space required for ordinary DAWGs and suffix trees. At the same time, the reduction of the number of states (2/3 less) and of transitions (about half less) makes the applications run faster. Time and space are saved simultaneously. The memory space used by our implementation of compact DAWGs requires about $6n$ integers for a word of length n. This is to be compared with $7n$ for DAWGs, $8n$ for suffix trees. It is just $2n$ for suffix arrays, but this is paid by a slower access to subwords.

This article is organized as follows. In Section 2 we recall the basic notions on DAWGs. Section 3 introduces the compact DAWG, also called compact suffix automaton, and contains the bounds on its size. We show in Section 3.4 how to build the compact DAWG from the DAWG in linear time with respect to the size of this latter structure. Direct construction algorithm for the compact DAWG is given in Section 4.

2 Definitions

Let Σ be a nonempty alphabet and Σ^* the set of words over Σ, with ε as the empty word. If w is a word in Σ^*, $|w|$ denotes its length, w_i its i^{th} letter, and $w_{i..j}$ its factor (subword) $w_i w_{i+1} \dots w_j$. If $w = xyz$ with $x, y, z \in \Sigma^*$, then x, y, and z are factors or subwords of w, x is a prefix of w, and z is a suffix of w. $S(x)$ denotes the set of all suffixes of x and $F(x)$ the set of its factors.

For an automaton, the tuple (p, a, q) denotes a transition of label a starting at p and ending at q. A roman letter is used for mono-letter transitions, a greek letter for multi-letter transitions. Moreover, $(p, \alpha]$ denotes a transition from p for which α is a prefix of its label. In this notation the target state is not given.

Here, we recall the definition of the DAWG, and a theorem about its implementation and size both proved in [3] and [9].

Definition 1. The Suffix Automaton of a word x, denoted $DAWG(x)$, is the minimal deterministic automaton (not necessarily complete) that accepts $S(x)$, the (finite) set of suffixes of x.

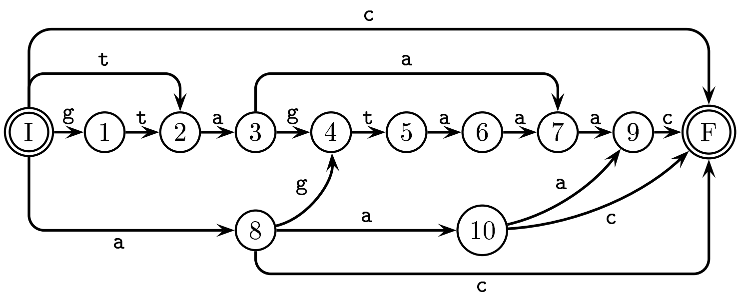

Fig. 1. $DAWG$(gtagtaaac).

For example, Figure 1 shows the DAWG of the word gtagtaaac. States that are double circled are terminal states.

Theorem 2. *The size of the DAWG of a word x is $\mathcal{O}(|x|)$ and the automaton can be computed in time $\mathcal{O}(|x|)$. The maximum number of states of the automaton is $2|x| - 1$, and the maximum number of edges is $3|x| - 4$.*

Recall that the right context (according to $S(x)$) of a factor u of x is $u^{-1}S(x)$. The syntactic congruence associated with $S(x)$ is denoted by $\equiv_{S(x)}$ and is defined, for $x, u, v \in \Sigma^*$, by:

$$u \equiv_{S(x)} v \iff u^{-1}S(x) = v^{-1}S(x).$$

We call *classes of factors* the congruence classes of the relation $\equiv_{S(x)}$. The longest word of a class of factors is called the *representative* of the class. States of $DAWG(x)$ are exactly the classes of the relation $\equiv_{S(x)}$. Since this automaton is not required to be complete, the class of words not occurring in x, corresponding to the empty right context, is not a state of $DAWG(x)$.

Among the congruence classes we make a selection of classes that are called *strict classes of factors* of $\equiv_{S(x)}$ and that are defined as follows.

Definition 3. Let u be a word of C, a class of factors of $\equiv_{S(x)}$. If at least two letters a and b of Σ exist such that ua and ub are factors of x, then C is called a **strict class of factors** of $\equiv_{S(x)}$.

We also introduce the function $endpos_x$: $F(x) \to \mathbb{N}$, defined, for a word u, by:

$$endpos_x(u) = \min\{|w| \mid w \text{ prefix of } x \text{ and } u \text{ suffix of } w\}$$

and the function $length_x$ defined on states of $DAWG(x)$ by:

$$length_x(p) = |u|, \text{ with } u \text{ representative of } p.$$

The word u also corresponds to the concatenated labels of transitions of the longest path from the initial state to p in $DAWG(x)$. Transitions that belong to the spanning tree of longest paths from the initial state are called *solid transitions*. Equivalently, for each transition (p, a, q) we have the property:

$$(p, a, q) \text{ is solid} \iff length_x(q) = length_x(p) + 1.$$

The function $length_x$ works as well for multi-letter transitions (transitions labeled by non-empty words), just replacing 1 in the above equivalence by the length of the label of the transition from p to q. This extends the notion of solid transitions to multi-letter transitions:

$$(p, \alpha, q) \text{ is solid} \iff length_x(q) = length_x(p) + |\alpha|.$$

In addition, we define the *suffix link* function on states of $DAWG(x)$ by the next statement.

Definition 4. Let p be a state of $DAWG(x)$, different from the initial state, and let u be a word of the equivalence class p. The **suffix link** of p, denoted by $s_x(p)$, is the state q which representative v is the longest suffix z of u such that $u \not\equiv_{S(x)} z$.

Note that, consequently to this definition, we have $length_x(q) < length_x(p)$. Then, by iteration, suffix links induce *suffix paths* in $DAWG(x)$, which is an important notion used by the construction algorithm. Indeed, as a consequence of the above inequality, the sequence $(p, s_x(p), s_x^2(p), ...)$ is finite and ends at the initial state of $DAWG(x)$. This sequence is called the *suffix path of* p.

3 Compact Directed Acyclic Word Graphs

3.1 Definition

Compaction of DAWGs is based on the deletion of some states and their outgoing transitions. This is possible by using multi-letter transitions and selecting strict classes of factors defined in the previous section (Definition 3).

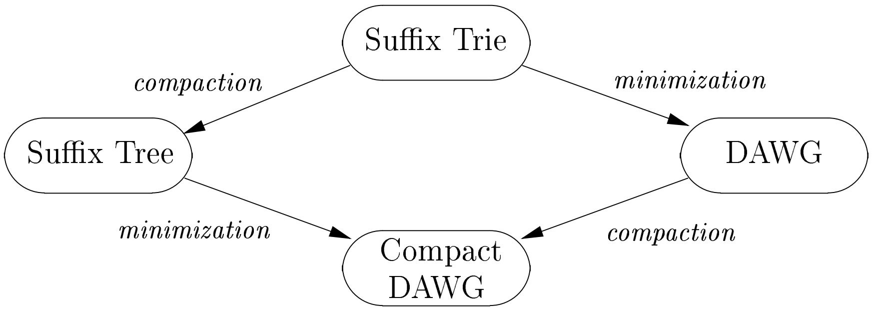

Fig. 2. Consider a word that has an end-marker. Its suffix tree is the compact version of the digital trie of its suffixes. Its DAWG is the minimized (in the sense of automata theory) version of the trie. The compact DAWG can be obtained either by minimizing the suffix tree of the word or by compacting its DAWG.

The definition of CDAWGs parallels the definition of suffix trees obtained from ordinary digital tries of all suffixes of a word. Indeed, disregarding how the end-marker required by suffix trees is managed, the CDAWG may be viewed as well as a compact version of the DAWG or as a minimized (in the sense of automata theory) version of the suffix tree (see Figure 2).

The compact DAWG is defined as follows.

Definition 5. The **Compact Directed Acyclic Word Graph** of a word x, denoted by $CDAWG(x)$, is the compaction of $DAWG(x)$ obtained by keeping only states that are either terminal states or strict classes of factors according to $\equiv_{S(x)}$, and by labeling transitions accordingly.

Consequently to Definition 3, strict classes of factors correspond to states that have an outdegree greater than one. So, we can delete every state having outdegree one exactly, except terminal states. Note that initial and final states are terminal states, so they are not deleted. An example of CDAWG is displayed in Figure 3.

The construction of the DAWG of a word containing repetitions shows that many states have outdegree one only. For example, in Figure 1, the DAWG of the word `gtagtaaac` has 12 states, 7 of which have outdegree one; it has 18

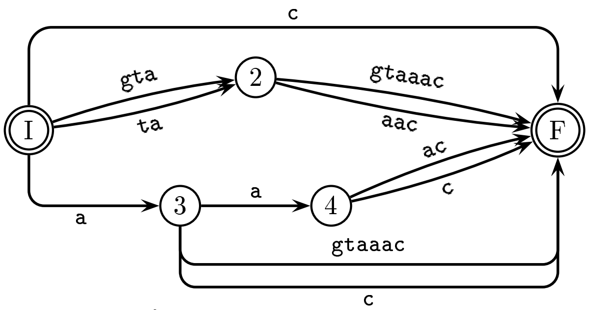

Fig. 3. *CDAWG*(`gtagtaaac`).

transitions. Figure 3 displays the compacted version, obtained after deletion of the 7 states, using multi-letter transitions. The resulting automaton has only 5 states and 11 edges.

According to experiments made on biological DNA sequences, considering them as words over the alphabet $\Sigma = \{a, c, g, t\}$, we got that more than 60% of states have outdegree one. So, the deletion of these states is worth, it provides an important saving. The average analysis of the number of states and edges in done in [5] in a Bernouilly model of probability.

When a state p is deleted, the deletion of its outgoing edges is realized by concatenating their label to the labels of incoming edges. For example, let r and p be states linked by a transition (r, b, p). The edges (r, b, p) and (p, a, q) are replaced by the edge (r, ba, q) if p is deleted. By recursion, this extends to every multi-letter transition (r, α, p).

In the example of Figure 3, one can note that, inside the word `gtagtaaac`, occurrences of `g` are followed by `ta`, and those of `t` and `gt` by `a`. The word `gta` is the representative of state 2, and there is no state corresponding to subwords `g`, `gt`, nor `t`. State I is directly connected to state 2 by edges (I,`gta`,2) and (I,`ta`,2). States 1 and 2 of Figure 1 no longer exist.

The suffix links defined on states of DAWGs remain valid when we reduce them to CDAWGs due to the next lemma, which proof is straightforward.

Lemma 6. *If p is a state of CDAWG(x), then $s_x(p)$ is a state of CDAWG(x).*

3.2 Size bounds

By Theorem 2 *DAWG*(x) is linear in $|x|$. As we shall see below (Section 3.3), labels of multi-letter transitions are implemented in constant space. So, the size of *CDAWG*(x) is also $\mathcal{O}(|x|)$. Meanwhile, as we delete many states and edges, we review the exact bounds on the number of states and edges of *CDAWG*(x). They are respectively denoted by *States*(x) and *Edges*(x).

Lemma 7. *Given $x \in \Sigma^*$, if $|x| = 0$, then States$(x) = 1$; if $|x| = 1$, then States$(x) = 2$; otherwise $|x| \geq 2$ and $2 \leq$ States$(x) \leq |x| + 1$.*

The upper bound on the number of states is reached when x is in the form $a^{|x|}$, for $a \in \Sigma$.

Proof. For $|x| \leq 1$, this is a mere verification. Assume now $|x| \geq 2$.

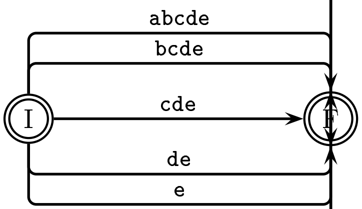

Fig. 4. A CDAWG with the minimum number of states, *CDAWG*(`abcde`).

The lower bound is obvious and obtained when x is composed of pairwise different letters.

Consider the suffix tree of $x\$$, where \$ is a marker. It has exactly $|x|+1$ leaves and at most $|x|$ internal nodes. Its minimization into $\mathit{CDAWG}(x)$ compacts all leaves into the final state F, and possibly put together other nodes. Removing the marker does not change the number of states. So, we have $\mathit{States}(x) \leq |x|+1$.

The word $a^{|x|}$ satisfies this property since each suffix $a^{|0|}$, $a^{|1|}$, ..., $a^{|x|}$ represents exactly one class. So, we have $|x|+1$ classes and the same number of states.

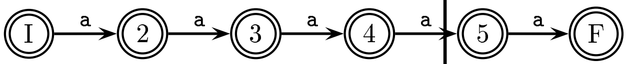

Fig. 5. A CDAWG with the maximum number of states, *CDAWG*(`aaaaa`).

Figures 4 and 5 display CDAWGs whose numbers of states are minimum and maximum, respectively, for words of length 5.

Lemma 8. *Given $x \in \Sigma^*$, if $|x| = 0$, $\mathit{Edges}(x) = 0$; if $|x| = 1$, $\mathit{Edges}(x) = 1$; otherwise $|x| \geq 2$ and $\mathit{Edges}(x) \leq 2|x| - 2$.*

The upper bound on the number of edges is reached when x is in the form $a^{|x|-1}c$, for a and c two different letters of Σ.

Proof. For $|x| \leq 1$, this is a mere verification. Assume now $|x| \geq 2$.

If x is in the form $a^{|x|}$, the number of edges is exactly $|x|$. So, we have to prove the upper bound for a word x containing at least two different letters. Consider the suffix tree of $x\$$. It has exactly $|x|+1$ leaves. It has at most $|x|-1$ internal nodes in this situation (because the root has outdegree 3). The number of edges in the tree is at most $2|x|-1$. After minimization into $\mathit{CDAWG}(x)$ and removing the marker, all edges may remain except the edge labeled by \$. This give the upper bound of $2|x| - 2$.

The automaton $CDAWG(a^{|x|-1}c)$, for a and c two different letters of Σ, has $|x|$ states and exactly $2|x| - 2$ edges, distributed as $|x| - 1$ solid edges and $|x| - 1$ non-solid edges.

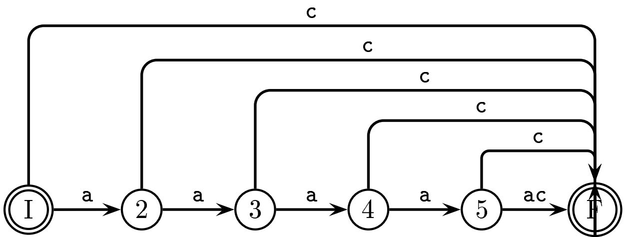

Fig. 6. A CDAWG with the maximum number of edges, $CDAWG(\texttt{aaaaac})$.

Figure 6 displays a *CDAWG* having the maximum number of edges for a word of length 6.

3.3 Implementation and experiments

Transition matrices and adjacency lists are two classical implementations of automata. The first one gives a direct access to transitions, but the memory space required is $\mathcal{O}(States(x) \times \text{card}(\Sigma))$. The second implementation stores only the exact number of transitions in memory, but needs $\mathcal{O}(\log \text{ card}(\Sigma))$ time to access them with standard searching techniques. When the size of the alphabet is great and the transition matrix is sparse, adjacency lists are obviously preferable. Otherwise, like for genomic sequences, transition matrix is a better choice, as shown by the experiments below. So, we only consider here transition matrices to implement CDAWGs.

We now describe the exact implementation of states and edges. We do this on a four-letter alphabet, so characters take 0.25 byte. We use integers encoded with 4 bytes. For each state, to encode the target state of outgoing edges, transitions matrices need a vector of 4 integers. Adjacency lists need, for each edge, 2 integers, one for the target state and another one for the pointer to the next edge.

The basic information required to construct the DAWG is composed of a table to implement the function s_x and one boolean value (0.125 byte) for each edge to know if it is solid or not. For the CDAWG, in order to implement multi-letter transitions, we need one integer for the $endpos_x$ value of each state, and another integer for the label length of each edge. And that is all.

Indeed, we can find the label of a transition by cutting off the length of this transition from the $endpos_x$ value of its target state. Then, we get both the

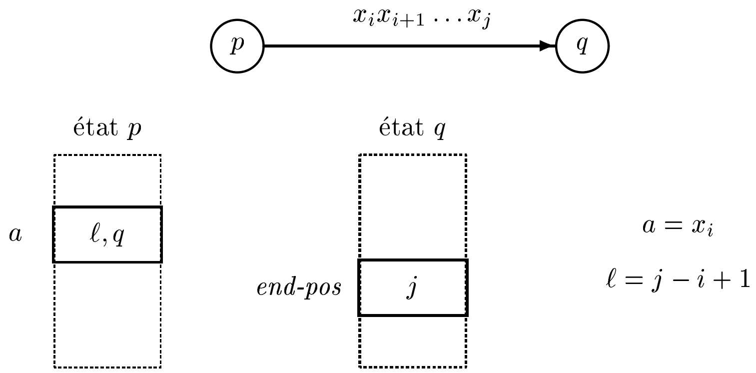

Fig. 7. Implementation of states and arcs in CDAWGs.

position of the label in the source and its length. Figure 7 illustrates this implementation. Keeping the source in memory is negligible considering the global size of the automaton (0.25 byte by character). This is quite a convenient solution also used for suffix trees.

Then, respectively for transitions matrices and adjacency lists, each state requires 20.5 and 17.13 bytes for the DAWG, and 40.5 and 41.21 bytes for the CDAWG. As a reference, suffix trees, as implemented by McCreight [18], need 28.25 and 20.25 bytes per state. Moreover, for CDAWG and suffix trees the source has to be stored in main memory. Theoretical average numbers of states, calculated by Blumer *et al.* ([5]), are $0.54n$ for CDAWG, $1.62n$ for DAWG, and $1,62n$ for suffix trees, when n is the length of x. This gives respective sizes in bytes per character of the source: 45.68 and 32.70 for suffix trees, 33.26 and 27.80 for DAWGs, and 22.40 and 22.78 for CDAWGs.

Considering the complete data structures required for applications, the function $endpos_x$ has to be added for the DAWG and the Suffix Tree. In addition, the occurrence number of each factor has to be stored in each state for all the structures. Therefore, the respective sizes in bytes per character of the source become : 58.66 and 45.68 for suffix trees, 46.24 and 40.78 for DAWGs, and 24.26 and 24.72 for CDAWGs.

Table 1 compares the sizes of implementations of DAWGs and CDAWGs meant for applications to DNA sequences. Sizes for random words of different lengths on a four-letter alphabet are also given. DNA sequences are *Saccharomyces cerevisiae* yeast chromosome II (chro II), a contig of *Escherichia Coli* DNA sequence (coli), and contigs 1 and 115 of *Bacillus Subtilis* DNA sequence (bs). Number of states and edges according to the length of the source and the memory space gain are displayed. Theoretical average ratios are given, computed from [5]. First, we observe there are 2/3 less states in the CDAWG, and near of half edges. Second, the memory space saving is about 50%. Third, the num-

Source x	$\|x\|$	$\frac{Nb\ states}{\|x\|}$		$\frac{Nb\ transitions}{\|x\|}$		$\frac{Nb\ transitions}{Nb\ states}$		memory gain
		dawg	cdawg	dawg	cdawg	dawg	cdawg	
chro II	807188	1,64	0,54	2,54	1,44	1,55	2,66	50,36%
coli	499951	1,64	0,54	2,54	1,44	1,53	2,66	51,95%
bs 1	183313	1,66	0,50	2,50	1,34	1,50	2,66	54,78%
bs 115	49951	1,64	0,54	2,54	1,44	1,55	2,66	50,16%
random	500000	1,62	0,55	2,54	1,47	1,57	2,68	49,53%
random	100000	1,62	0,55	2,55	1,47	1,57	2,68	49,35%
random	50000	1,62	0,54	2,54	1,46	1,56	2,68	49,68%
random	10000	1,62	0,54	2,54	1,46	1,56	2,68	49,47%
theor. aver. ratios		**1,63**	**0,54**	**2,54**	**1,46**	**1,56**	**2,67**	**50,55%**

Table 1. Statistics on the sizes of real DAWGs and CDAWGs.

ber of edges per state is going up to 2.66 when considering CDAWGs. With a four-letter alphabet, this is interesting to note because the implementation by transition matrix requires less space than an implementation by adjacency lists. At the same time, this keeps a direct access to transitions.

3.4 Constructing CDAWGs from DAWGs

The DAWG construction is fully exposed and demonstrated in [3], [9] and [11]. As we show in this section, the CDAWG is easily derived from the DAWG.

Indeed, we just need to apply the definition of the CDAWG. The computation is done by the function *Reduction* below. Observe that, in this function, $state(p, a]$ denotes the target state of the transition $(p, a]$. The computation is done during a depth-first traversal of the automaton, and runs in time linear in the number of transitions of $DAWG(x)$. Then, by theorem 2, the computation runs in time linear in the length of the text.

The main drawback of this construction of CDAWGs is that it requires the previous construction of DAWGs. Therefore, the overall construction takes time and memory space proportional to $DAWG(x)$, though $CDAWG(x)$ is significantly smaller. So, it is better to construct the CDAWG directly.

Reduction (state E) returns (ending state, length of redirected edge)
If (E not marked) **Then**
  For all existing edge $(E, a]$ **Do**
    $(state(E, a]\ ,\ |label((E, a])|) \leftarrow$ Reduction(ıtastate$(E, a]$);
  mark$(E) \leftarrow$ TRUE;
If (E is of outdegree one) **Then**
  Let $(E, a]$ this edge ;
  Return $(state(E, a]\ ,\ 1 + |label((E, a])|)$;
Else
  Return (E,1);

4 Direct Construction of CDAWG

In this section, we give the direct construction of CDAWGs. The running time of the algorithm is linear in the size of the input word x on a fixed alphabet. The memory space is proportional to the size of the automaton, and consequently is also linear by Lemmas 7 and 8.

4.1 Algorithm

Since the CDAWG of x is a minimization of its suffix tree, it is rather natural to base the direct construction on McCreight's algorithm [18]. Meanwhile, properties of the DAWG construction are also used, especially the suffix link function (notion that is different from the suffix links of McCreight's algorithm), lengths of longest paths, and positions, as explained in the previous section.

First, we introduce the notions used by the algorithm, some of them are taken from [18]. The algorithm constructs the CDAWG of the word x of length n, noted $x_{0..n-1}$. The automaton is defined by a set of states and transitions, where I and F denotes the initial and the final states respectively. A *partial path* represents a connected sequence of edges between two states of the automaton. A *path* is a partial path that begins at I. The label of a path is the concatenation of the labels of corresponding edges.

The *locus*, or *exact locus*, of a string is the end of the path labeled by the string. The *contracted locus* of a string α is the locus of the longest prefix of α whose locus is defined. -

Preliminary Algorithm Basically, the algorithm that builds *CDAWG*(x) inserts into the current automaton the paths corresponding to all the suffixes of x, from the longest to the shortest suffix. We define suf_i as the suffix $x_{i..n-1}$ of x. We denote by $\mathcal{A}_i$ the automaton constructed after the insertion of all the suf_j for $0 \leq j \leq i$.

Figure 8 displays six steps during the construction of *CDAWG*(`aabbabbc`). In this figure (and the following), the dashed edges represent suffix links, links that are defined on states and that are used in the next section.

At the beginning of the algorithm the automaton is initialized with the two states I and F only. At step i ($i > 0$), the algorithm inserts a path corresponding to suf_i into $\mathcal{A}_{i-1}$ and produces $\mathcal{A}_i$. The main loop of the algorithm satisfies the following invariant properties:

P1: at the beginning of step i, all suffixes suf_j, $0 \leq j < i$, are paths in $\mathcal{A}_{i-1}$.
P2: at the beginning of step i, the states of $\mathcal{A}_{i-1}$ are in one-to-one correspondence with the longest common prefixes of pairs of suffixes longer than suf_j.

We define $head_i$ as the longest prefix of suf_i which is also a prefix of suf_j for some $j < i$. Equivalently, $head_i$ is the longest prefix of suf_i that is also label of a path in $\mathcal{A}_{i-1}$. We define $tail_i$ as $head_i^{-1} suf_i$.

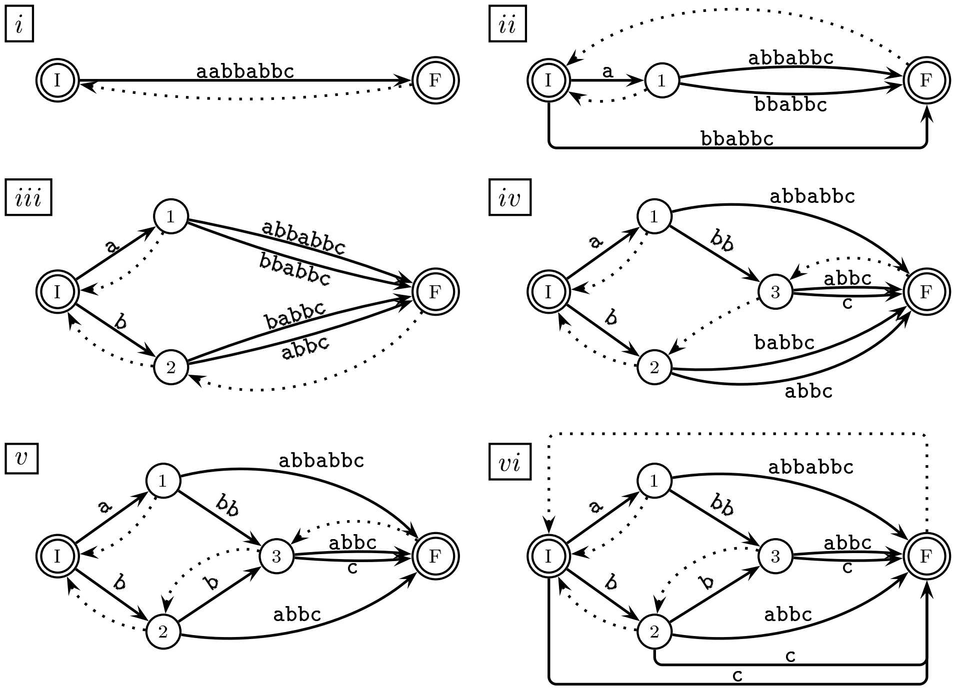

Fig. 8. Six steps during the construction of $CDAWG($`aabbabbc`$)$. The pictures display the situation after the insertion of suf_0=`aabbabbc` (i), suf_2=`bbabbc` (ii), suf_3=`babbc` (iii), suf_4=`abbc` (iv), and suf_5=`bbc` (v). vi shows the final automaton.

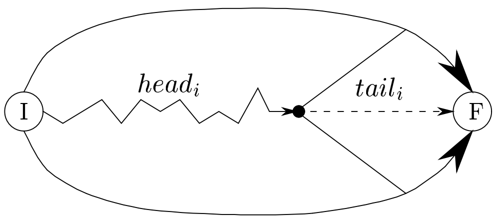

Fig. 9. Scheme of the insertion of suf_i in $\mathcal{A}_{i-1}$: there already is a path labeled by the prefix $head_i$ of suf_i.

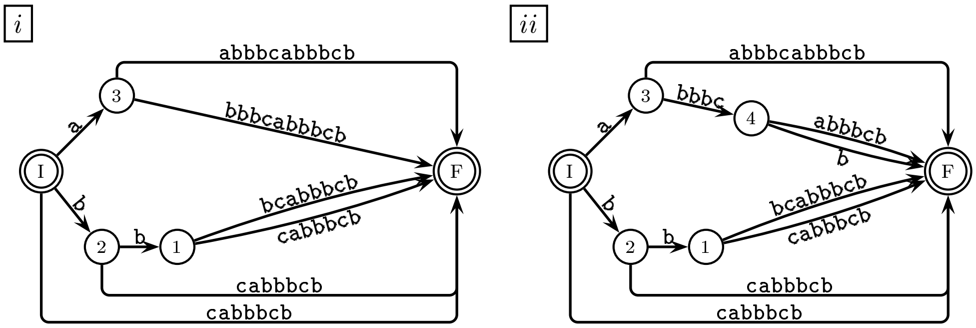

Fig. 10. Example of the execution of *SlowFind* during the construction of *CDAWG*(aabbbcabbbcb). For the insertion of suf_6=abbbcb, we have $head_6$=abbbc. Since the path labeled by abbbc ends in the middle of the edge (3,bbbcabbbcb,F), state 4 is created, splitting the edge into (3,bbbc,4) and (4,abbbcb,F). A new edge is created, (4,b,F).

At step i, the preliminary algorithm has to insert $tail_i$ from the locus of $head_i$ into $\mathcal{A}_{i-1}$ (see Figure 9). To do so, the contracted locus of $head_i$ in $\mathcal{A}_{i-1}$ is found with the help of function *SlowFind* that compares letter-to-letter the right path of $\mathcal{A}_{i-1}$ to suf_i. An example of execution of this function is shown in Figure 10. This part is similar to the corresponding McCreight's procedure, except on a point discussed below (redirection of edges). If there is a state at the end of the path, it is the locus of $head_i$. Otherwise it is created at the middle of the last encountered edge by splitting it. In any case, an edge labeled by $tail_i$ is created from the locus of $head_i$ to F. The preliminary algorithm is given below.

Preliminary Algorithm
1. **For all** suf_i ($i \in$[0..n-1]) **Do**
2. $(q, \gamma) \leftarrow$ *SlowFind*(I);
3. **If** $(\gamma = \varepsilon)$ **Then**
4. insert (q,$tail_i$,F);
5. **Else**
6. create v locus of $head_i$ splitting $(q, \gamma]$
and insert (v,$tail_i$,F);
or redirect $(q, \gamma]$ onto v,
the last created state;
7. **End For all**;
8. mark terminal states;

The function *SlowFind* returns a pair (q, γ) such that q is the last encountered state on the path $head_i$, state that is the representative of $head_i\gamma^{-1}$. This keeps accessible the transition that may be split if the state q is not the exact locus of $head_i$, *i.e.* if $\gamma \neq \varepsilon$.

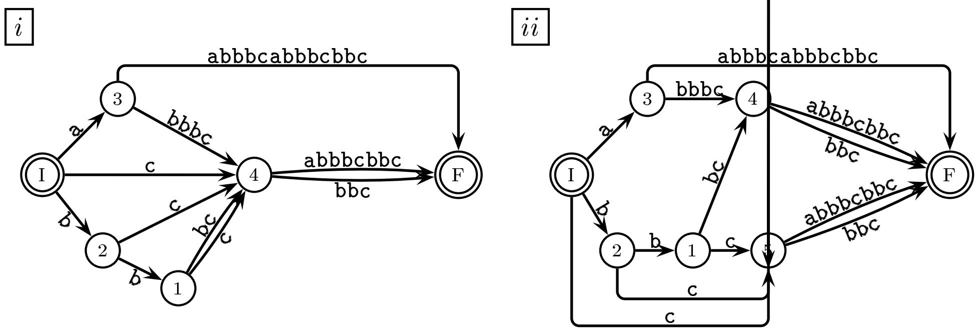

Fig. 11. Example of a duplication in *SlowFind* during the construction of *CDAWG*(aabbbcabbbcbbc). The insertion of suf_{11}=bbc leads to state 4. As the last edge (1,c,4) is non-solid (i), state 4 is cloned into state 5 (ii), and the edge (1,c,4) becomes (1,c,5).

If a non-solid edge is encountered during the execution of *SlowFind*, its target state has to be duplicated in a clone and the non-solid edge is redirected to this clone. The redirected transition becomes solid. An example of duplication is given in Figure 11.

In some situation, an edge can be redirected. This happens when a state has just been created at the previous step. The edge is redirected to this state and its label is updated accordingly. Such a situation appears in Figure 8 (case v) for the construction of *CDAWG*(aabbabbc) : the insertion of suf_5=bbc induces the redirection of the edge (2,babbc,F), which becomes (2,b,3). In the above situation, the suffix link of the last created state is unknown during the insertion of the current suffix. And the redirections go on until the suffix link is found.

Finally, when $tail_i = \varepsilon$ at the end of the construction, terminal states are marked along the suffix path of F.

From the above discussion, a proof of the invariance of properties P1 and P2 can be derived. Thus, at the end of the algorithm all subwords of x and only these words are labels of paths in the automaton (property P1). By property P2, states correspond to strict classes of factors (when the longest common prefix of a pair of suffixes is not equal to any of them) or to terminal states (when the contrary holds). This gives a sketch of the correctness of the algorithm.

The running time of the preliminary algorithm is $\mathcal{O}(|x|^2)$ (with an implementation by transition matrix), like is the sum of lengths of all suffixes of the word x.

Linear Algorithm To get a linear-time algorithm, we use together properties of DAWGs construction and of suffix trees construction. The main feature is the notion of suffix links. They are defined as for DAWGs in Section 2, definition that remains valid by Lemma 6. They are the clue for the linear running time of the algorithm.

Three elements have to be pointed out about suffix links in the CDAWG. First, we do not need to initialize suffix links. Indeed, when suf_0 is inserted, x_0 is obviously a new letter because no letter of x has been scanned so far, which directly induces s_x(F)=I. Note that s_x(I) is never used, and so never defined. Second, traveling along the suffix path of a state p does not necessarily end at state I. Indeed, with multi-letter transitions, if $s_x(p)$=I we have to treat the suffix $a^{-1}\alpha$ ($a \in \Sigma$) where α is the representative of p. And third, suffix links induce the following invariant property satisfied at step i:

P3: at the beginning of step i, the suffix links are defined for each state of $\mathcal{A}_{i-1}$ according to Definition 4, except maybe for the lastly-created state.

The next remark allows redirections without having to search with *SlowFind* for existing states belonging to a same class of factors.

Remark. Let $\alpha\beta$ have locus p and assume that $q = s_x(p)$ is the locus of β. Then, p is the locus of suffixes of $\alpha\beta$ whose lengths are greater than $|\beta|$.

The algorithm has to deal with suffix links each time a state is created. This happens when a state is duplicated, as illustrated by Figure 11, and when a state is created after the execution of *SlowFind*.

During a duplication, suffix links are updated as follows. Let w be the clone of q. In regard to strict classes of factors and Definition 4, the class of w is inserted "between" the ones of q and $s_x(q)$. So, we update suffix links by setting $s_x(w) = s_x(q)$ and then $s_x(q) = w$.

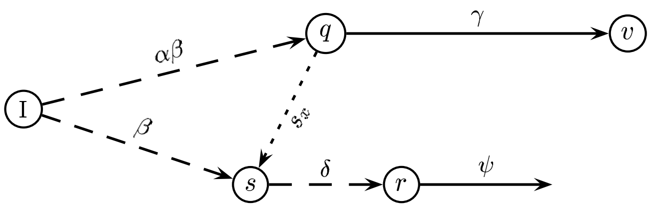

Fig. 12. Searching for $s_x(v)$ using a suffix link.

After the execution of *SlowFind*, if state v is created, we have to compute its suffix link, $s_x(v)$. Let γ be the label of the transition starting at q and ending at v. To compute the suffix link of v, the algorithm goes through the path having label γ from the suffix link of q, $s = s_x(q)$. The operation is repeated if necessary. Figure 12 displays a scheme of this search. The thick dashed edges represent paths in the automaton, and the thin dashed edge represents the suffix link from q to s. The search, as for the duplication, realizes the insertion of a series of suffixes. To travel along the path, we use the function *FastFind*, similar to the one used in McCreight's algorithm [18], that goes through transitions comparing just the first letters of their labels. This function returns the last encountered state and edge.

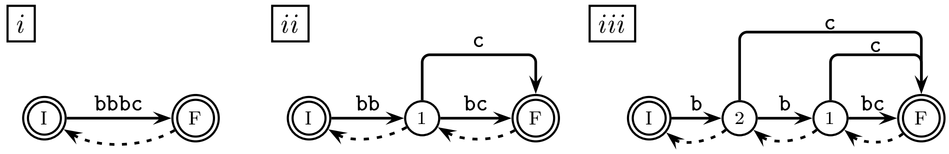

Fig. 13. Example of execution of *FastFind* ending with a solid edge during the construction of *CDAWG*(bbbc). The insertion of suf_1=bbc leads to create state 1. Then *FastFind* works from I with path b. This leads to the middle of the edge (I,bb,1) (*ii*) that is solid. Since we cannot redirect this edge, state 2 is created, splitting (I,bb,1) into (I,b,2) and (2,b,1) (*iii*). The edge (2,c,F) is added, $s_x(1)$ is set to 2, and $s_x(2)$ is set to I.

Let r and $(r, \psi]$ be the state and transition returned by *FastFind*. If r is the exact locus of γ, it is the wanted state, and we set then $s_x(v) = r$. Else, if $(r, \psi]$ is a solid edge, then a new node w is created. The edge $(r, \psi]$ is split, its initial part becomes (r, ψ, w), and the transition (w,$tail_i$,F) is added. Such an example is displayed in Figure 13.

The last situation to consider is when $(r, \psi]$ is non-solid. Then, the edge is replaced by (r, ψ, v). Such an example is displayed in Figure 14.

In the two last cases, since $s_x(v)$ is not found, we run *FastFind* again with $s_x(r)$ and ψ, and this goes on until $s_x(v)$ is eventually found, that is, when $\psi = \varepsilon$.

FastFind is used in the same manner when a state is created by duplication during the execution of *SlowFind*.

The discussion shows how suffix links are updated to insure that property P3 is satisfied. The operations do not influence the correctness of the algorithm, sketched in the last section, but yield the following linear-time algorithm. Its time complexity is discussed in the next section.

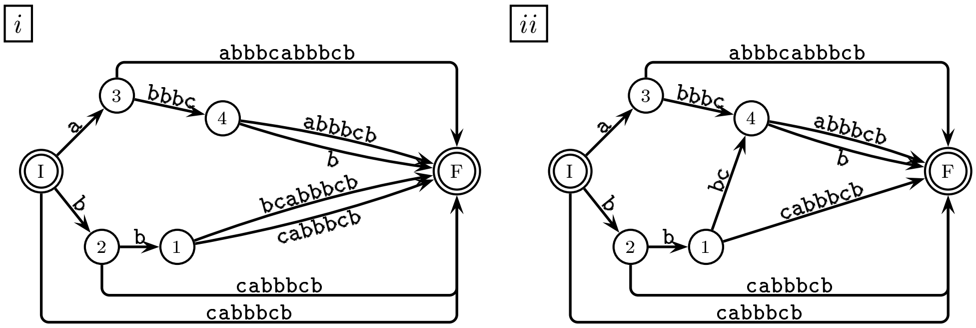

Fig. 14. Example of execution of *FastFind* ending with a non-solid edge during the construction of *CDAWG*(aabbbcabbbcb). When suf_6=abbbcb is inserted and state 4 created, we have to look for $s_x(4)$. As $s_x(3)$=I, we travel along edges from I to find the end of the path labeled by bbbc with *FastFind*. As this path ends in the middle of the non-solid edge (1,bcabbbcb,F), this one is replaced by (1,bc,4). Then, *FastFind* runs again from state 2 with the word bc, in order to eventually find $s_x(4)$.

```
Linear Algorithm
p ← I;   i ← 0;
While not end of x Do
      (q, γ) ← SlowFind(p);
      If (γ = ε) Then
            insert (q,tail_i,F);
            s_x(F) ← q;
            If (q ≠ I) Then p ← s_x(q) Else p ← I;
      Else
            create v locus of head_i splitting (q, γ];
            insert (v,tail_i,F);
            s_x(F) ← v;
            find r = s_x(v) with FastFind;
            p ← r;
      update i;
End While;
mark terminal states;
```

4.2 Complexity

Theorem 9. *The algorithm that builds the CDAWG of a word x of Σ^* can be implemented in time $\mathcal{O}(|x|)$ and space $\mathcal{O}(|x| \times \mathrm{card}(\Sigma))$ with a transition matrix, or in time $\mathcal{O}(|x| \times \log\ \mathrm{card}(\Sigma))$ and space $\mathcal{O}(|x|)$ with adjacency lists.*

Proof. As recalled in section 3.1, the size of *CDAWG*(x) is linear in the length of x, both in term of number of states and number of edges. Tables $endpos_x$,

$length_x$ and s_x take $\mathcal{O}(States(x))$ space. So, an implementation by transition matrix takes $\mathcal{O}(|x| \times \text{card}(\Sigma))$ space. By adjacency lists, it takes $\mathcal{O}(|x|)$ space.

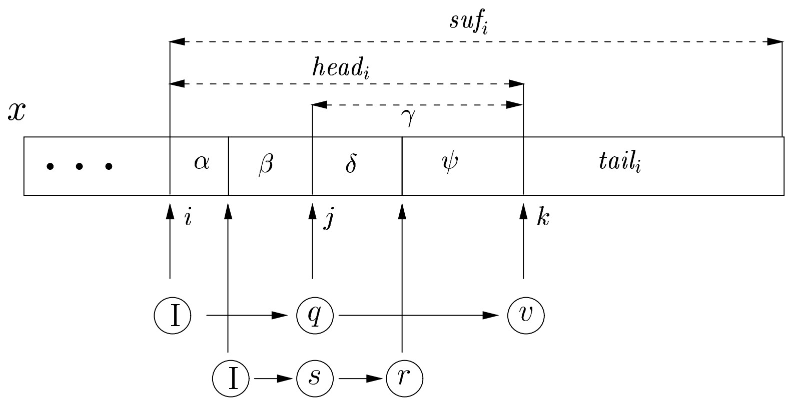

Fig. 15. Positions of labels when suf_i is inserted. States I,q,v represent the scheme of *SlowFind* and states I,s,r represent the scheme of searching for $s_x(q)$, as in Figure 12.

The complexity of the algorithm essentially depends on the number of branchings made on states of the automaton. We prove that this number is linear, which implies the running times of the statement: $\mathcal{O}(|x|)$ with a transition matrix and $\mathcal{O}(|x| \times \log\ \text{card}(\Sigma))$ with adjacency lists.

Branchings during the execution of the algorithm are done during calls to *SlowFind* and *FastFind*. The generic situation is displayed in Figure 15. When *SlowFind* operates, the current letter of x, pointed by k, is compared with a letter of the label of an edge. Doing so, k is strictly incremented, and never after decremented. During calls to *FastFind*, each letter comparison increases strictly the value of j, value that never decreases hereafter. This shows that the number of branchings is linear.

This ends the sketch of the proof.

5 Conclusion

We have considered the Compact Direct Acyclic Word Graph, which is an efficient compact data structure to represent all subwords, or factors, of a word. There are several data structures used to store this set. The present structure provides an interesting space gain compared to the standard DAWG, and also when compared with suffix trees. From the theoretical point of view, the upper bounds are of $|x| + 1$ states and $2|x| - 2$ transitions. This saves $|x|$ states and $|x|$ transitions of the DAWG and at the same time leads to a faster use. From the practical point of view, experiments on genomic DNA sequences and on random strings display a memory space gain of 50% with respect to the DAWG. Moreover, when the size of the alphabet is small, transition matrices do not take more space than adjacency lists, keeping direct access to transitions. Thus, we can construct the data structure of twice larger strings, keeping them in main memory, which is actually important to get efficient treatments.

This work shows that the CDAWG can be constructed directly. The algorithm is linear in the length of the text (on a fixed alphabet). Of course, it is simpler to compute, by reduction, the CDAWG from the DAWG. But the present algorithm saves time and space simultaneously.

References

1. A. Apostolico. The myriad virtues of subword trees. In A. Apostolico & Z. Galil, editor, *Combinatorial Algorithms on Words.*, pages 85–95. Springer-Verlag, 1985.
2. A. Blumer, J. Blumer, A. Ehrenfeucht, D. Haussler, and R. McConnel. Linear size finite automata for the set of all subwords of a word: an outline of results. *Bull. European Assoc. Theoret. Comput. Sci.*, 21:12–20, 1983.
3. A. Blumer, J. Blumer, D. Haussler, A. Ehrenfeucht, M.T. Chen, and J. Seiferas. The smallest automaton recognizing the subwords of a text. *Theoret. Comput. Sci.*, 40:31–55, 1985.
4. A. Blumer, J. Blumer, D. Haussler, and R. McConnell. Complete inverted files for efficient text retrieval and analysis. *Journal of the Association for Computing Machinery*, 34(3):578–595, July 1987.
5. A. Blumer, D. Haussler, and A. Ehrenfeucht. Average sizes of suffix trees and dawgs. *Discrete Applied Mathematics*, 24:37–45, 1989.
6. B. Clift, D. Haussler, R. McDonnell, T.D. Schneider, and G.D. Stormo. Sequence landscapes. *Nucleic Acids Research*, 4(1):141–158, 1986.
7. M. Crochemore. Recherche linéaire d'un carré dans un mot. *C. R. Acad. Sci. Paris Sér. I Math.*, 296:781–784, 1983.
8. M. Crochemore. Optimal factor tranducers. In A. Apostolico and Z. Galil, editors, *Combinatorial Algorithms on Words*, volume 12 of *NATO Advanced Science Institutes, Series F*, pages 31–44. Springer-Verlag, Berlin, 1985.
9. M. Crochemore. Transducers and repetitions. *Theoret. Comput. Sci.*, 45(1):63–86, 1986.
10. M. Crochemore. Longest common factor of two words. In H. Ehrig, R. Kowalski, G. Levi, and U. Montanari, editors, *TAPSOFT*, number 249 in Lecture Notes in Computer Science, pages 26–36. Springer-Verlag, Berlin, 1987.
11. M. Crochemore and C. Hancart. Automata for matching patterns. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*. Springer-Verlag, 1997. to appear.
12. M. Crochemore and W. Rytter. *Text Algorithms*, chapter 5-6, pages 73–130. Oxford University Press, New York, 1994.
13. M. Farach. Optimal suffix tree construction with large alphabets. manuscript, October 1996.
14. R. W. Irving. Suffix binary search trees. *Technical report TR-1995-7, Computing Science Department, University of Glasgow*, April 1995.
15. J. Karkkainen. Suffix cactus : a cross between suffix tree and suffix array. *Combinatorial Pattern Matching*, 937:191–204, July 1995.

16. C. Lefevre and J-E. Ikeda. The position end-set tree: A small automaton for word recognition in biological sequences. *CABIOS*, 9(3):343–348, 1993.
17. U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, Oct. 1993.
18. E. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, Apr. 1976.
19. G. A. Stephen. *String searching algorithms*. World Scientific Press, 1994.
20. E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14:249–260, 1995.
21. E. Ukkonen and D. Wood. Approximate string matching with suffix automata. *Algorithmica*, 10(5):353–364, 1993.
22. P. Weiner. Linear pattern matching algorithm. In *14th Annual IEEE Symposium on Switching and Automata Theory*, pages 1–11, Washington, DC, 1973.