HAL
open science

Direct construction of compact Directed Acyclic Word Graphs

Maxime Crochemore, Renaud Vérin

To cite this version:

Maxime Crochemore, Renaud Vérin. Direct construction of compact Directed Acyclic Word Graphs. Combinatorial Pattern Matching (Aarhus, 1997), 1997, France. pp.116-129. hal-00620006

HAL Id: hal-00620006

https://hal.science/hal-00620006v1

Submitted on 13 Feb 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Direct construction of 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 an 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.

Keywords: pattern matching algorithm, suffix automaton, DAWG, Compact DAWG, suffix tree, index on text.

1 Introduction

In the classical string-matching problem for a word w and a text T, we want to know if w occurs in T, *i.e.*, if w is a factor of T. In many applications, the same text is queried several times. So, efficient solutions are based on data structures built on the text that serve as an index to look for any word w in T. The typical running of various implementations of the search is $\mathcal{O}(|w|)$ (on a fixed alphabet). Among the implementations, the *suffix tree* ([13]) is the most popular. Its size and construction time are linear in the length of the text. It has been studied and used extensively. Apostolico [2] lists over 40 references on it, and Manber and Myers [12] mention several others. Many variants have been developed, like *suffix arrays* [12], *PESTry* [11], *suffix cactus* [10], or *suffix binary search trees* [9]. Besides, the suffix trie, the non-compact version of the suffix tree, has been refined to the *suffix automaton* (*Directed Acyclic Word Graph, DAWG*). This automaton is a good alternative to represent the whole set of factors of a text. It is the minimal automaton accepting this set. It has been fully exposed by Blumer [3] and Crochemore [7]. As for the suffix tree, its construction and size is linear in the length of the text.

In the genome research field, DNA sequences can be viewed as words over the alphabet $\{a, c, g, t\}$. They become subjects for linguistic and statistic analysis. For this purpose, suffix automata are useful data structures. Indeed, the structure is fast to compute and easy to use.

Meanwhile, the length of sequences in databases grows rapidly and the bottleneck to using the above data structures is their size. Keeping the index in main

memory is more and more difficult for large sequences. So, having a structure using as little space as possible is appreciable for its construction as well as for its utilization. Compression methods are of no use to reduce the memory space of such indexes because they eliminate the direct access to substrings. On the contrary, the *Compact Directed Acyclic Word Graph* (CDAWG) keeps the direct access while requiring less memory space. The structure has been introduced by Blumer *et al.* [4, 5]). The automaton is based on the concatenation of factors issued from a same context. This concatenation induces the deletion of all states of outdegree one and of their corresponding transitions, excepting terminal states. This saves 50% of memory space. At the same time, the reduction of the number of states (2/3 less) and transitions (about half less) makes the applications run faster. Both time and space are saved.

In this paper, we give an algorithm to build compact DAWGs. This direct construction avoids constructing the DAWG first, which makes it suitable for the actual DNA sequences (more than 1.5 million nucleotides for some of them). The compact DAWG allows to apply standard treatment on sequences twice as long in reasonable time (a few minutes).

In Section 2 we recall the basic notions on DAWGs. Section 3 introduces the compact DAWG, also called compact suffix automaton, with the bounds on its size. We show in Section 4 how to build the CDAWG from the DAWG in time linear in the size of this latter structure. The direct construction algorithm for the CDAWG is given in Section 5. A conclusion follows.

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 ith letter, and $w_{i..j}$ its factor (subword) $w_i w_{i+1} \ldots w_j$. If $w = xyz$ with $x, y, z \in \Sigma^*$, then x, y, and z denote some 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.

Here, we recall the definition of the DAWG, and a theorem about its implementation and its size proved in [3] and [7].

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.

For example, Figure 1 shows the DAWG of the word `gtagtaaac`. States which 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 $3|x| - 4$.*

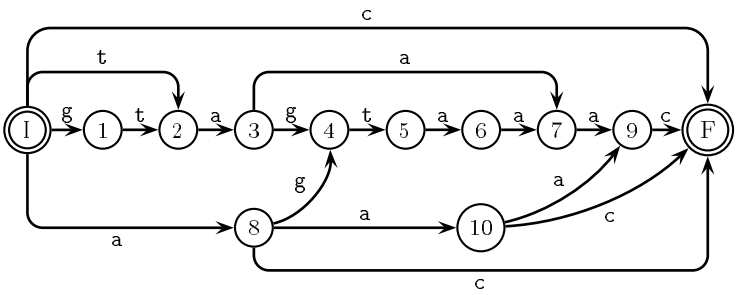

Fig. 1. $DAWG(\texttt{gtagtaaac})$

Recall that the right context of a factor u of x is $u^{-1}S(x)$. The syntactic congruence, denoted by $\equiv_{S(x)}$, associated with $S(x)$ 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 $DAWG(x)$.

Moreover, we induce a selection among the congruence classes that we call *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 we say that C is a **strict class of factors** of $\equiv_{S(x)}$.

We also introduce the function $endpos_x : F(x) \to \mathbb{N}$, defined, for every 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 $DAWG(x)$. The 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, just replacing 1 in the above equivalence by the length of the label of the transition. 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 addititon, we define the *suffix link* for a state of $DAWG(x)$ by:

Definition4. Let p be a state of $DAWG(x)$, different from the initial state, and let u 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

The compression of DAWGs is based on the deletion of some states and their corresponding transitions. This is possible using multi-letter transitions and the selection of strict classes of factors defined in the previous section (Definition 3). Thus, we define the Compact DAWG as follows.

Definition5. 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, the strict classes of factors correspond to the 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 too, so they are not deleted.

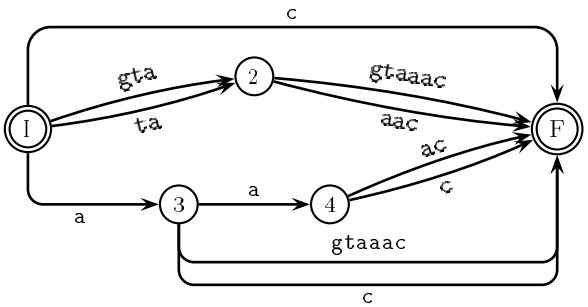

Fig. 2. $CDAWG$(gtagtaaac)

The construction of the DAWG of a word including some 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 transitions. Figure 2 displays the result after the deletion of these states, using multi-letter transitions. The resulting automaton has only 5 states and 11 edges.

According to experiments to construct DAWGs of biological DNA sequences, considering them as words over the alphabet $\Sigma = \{a, c, g, t\}$, we got that more than 60% of states have an 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 is done in [5] in a Bernouilly model of probability.

When a state p is deleted, the deletion of outgoing edges is realized by adding the label of the outgoing edge of the deleted state to the labels of its incoming edges. For example, let r, p and q be states linked by transitions (r, b, p) and (p, a, q). We replace the edges (r, b, p) and (p, a, q) by the edge (r, ba, q). By recursion, we extend this method to every multi-letter transition (r, α, p).

In the example (Figure 1), one can note that, inside the word `gtagtaaac`, occurrences of `g` are followed by `ta`, and those of `t` and `gt` by `a`. So, `gta` is the representative of state 3 and it is not necessary to create states for `g` and (`gt` or `t`). Then, we directly connect state I to state 3 with edges (I,`gta`,3) and (I,`ta`,3). States 1 and 2 are so deleted.

The suffix links defined on states of DAWGs remain valid when we reduce them to CDAWGs because of the next lemma.

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 $Edges(x)$.

Corollary 7. *Given $x \in \Sigma^*$, if $|x| = 0$, then $States(x) = 1$; if $|x| = 1$, then $States(x) = 2$; else $|x| \geq 2$, then $2 \leq States(x) \leq |x| + 1$ and the upper bound is reached when x is in the form $a^{|x|}$, where $a \in \Sigma$.*

Corollary 8. *Given $x \in \Sigma^*$, if $|x| = 0$, $Edges(x) = 0$; if $|x| = 1$, $Edges(x) = 1$; else $|x| \geq 2$, then $Edges(x) \leq 2|x| - 2$ and this upper bound is reached when x is in the form $a^{|x|-1}c$, where a and c are two different letters of Σ.*

3.3 Implementation and Results

Transition matrices and adjacency lists are the classical implementations of automata. Their principal difference lies in the implementation of transitions. The first one gives a direct access to transitions, but requires $\mathcal{O}(States(x) \times \text{card}(\Sigma))$. The second one stores only the exact number of transitions in memory, but needs $\mathcal{O}(\log \text{card}(\Sigma))$ time to access them. When the size of the alphabet is big and the transition matrix is sparse, adjacency lists are 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 ending state. Then, we got the position of the label in the source and its length. 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. Figure 3 displays how the

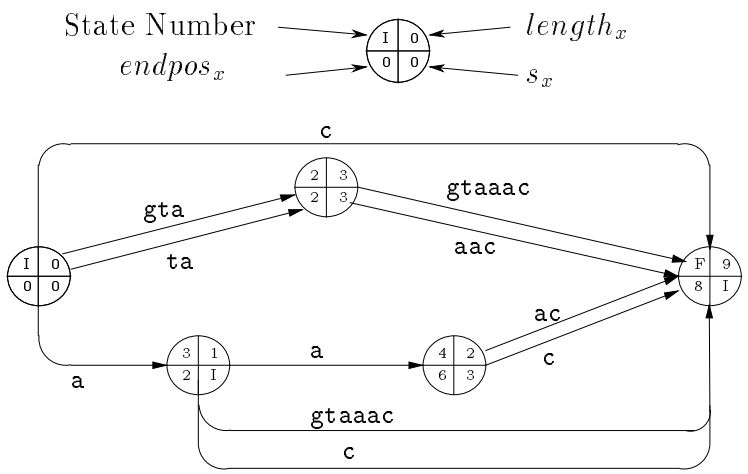

Fig. 3. Data Structure of *CDAWG*(gtagtaaac)

states of *CDAWG*(gtagtaaac) are implemented.

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 [13], 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.

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. Statistic table with account between DAWG and CDAWG.

Moreover, Table 1 compares sizes of DAWG and CDAWG meant for applications to DNA sequences. Sizes for random words of different lengths and $|\Sigma| = 4$ 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, calculated from Blumer *et al.* ([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 number of edges by state is going up to 2.66. With a four-letter alphabet, this is interesting because the transition matrix becomes smaller than adjacency lists. At the same time, we keep a direct access to transitions.

4 Constructing CDAWG from DAWG

The DAWG construction is fully exposed and demonstrated in [3] and [7]. 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 recursively. This is computed by the function *Reduction*, given below. Observe that, in this function, $state(p, a]$ denotes the state pointed to by the transition $(p, a]$. The computation is done with 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 also runs in time linear in the length of the text.

However, this method needs to construct the DAWG first, which spends 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)
1.  If (E not marked) Then
2.      For all existing edge (E, a] Do
3.          (state(E, a] , |label((E, a])|) ← Reduction(state(E, a]);
4.      mark(E) ← TRUE;
5.  If (E is of outdegree one) Then
6.      Let (E, a] this edge;
7.      Return (state(E, a] , 1 + |label((E, a])|);
8.  Else
9.      Return (E,1);
```

5 Direct Construction of CDAWG

In this section, we give the direct construction of CDAWGs and show that the running time is linear in the size of the input word x on a fixed alphabet.

5.1 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 [13]. Meanwhile, properties of the DAWG construction are also used, especially suffix links (notion that is different from the suffix links of McCreight's algorithm), lengths, and positions, as explained in the previous section.

First, we introduce the notions used by the algorithm, some of them are taken from [13]. 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, especially with I and F, the initial and final states. 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 to build CDAWG inserts the paths corresponding to all the suffixes of x from the longest to the shortest. 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 $0 \leq j \leq i$.

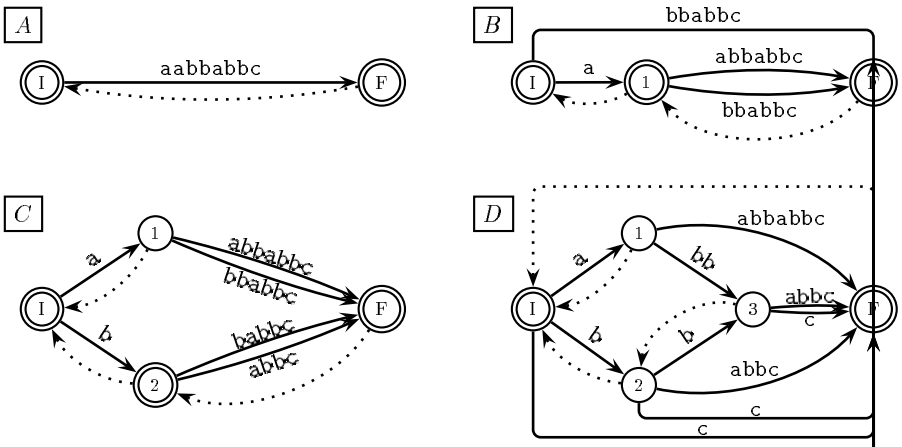

Fig. 4. Construction of $CDAWG$(aabbabbc)

Figure 4 displays four steps of the construction of $CDAWG$(aabbabbc). In this Figure (and the followings), the dashed edges represent suffix links of states, which are used subsequently. We initialize the automaton $\mathcal{A}_\varepsilon$ with states I and F. At step i ($i > 0$), the algorithm inserts a path corresponding to suf_i in $\mathcal{A}_{i-1}$ and produces $\mathcal{A}_i$. 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 which is also a path of $\mathcal{A}_{i-1}$. We define $tail_i$ as $head_i^{-1} suf_i$. At step i, the preliminary algorithm has to insert $tail_i$ from the locus of $head_i$ in $\mathcal{A}_{i-1}$ (see Figure 5).

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. This is similar to the corresponding McCreight's procedure, except on what is explained below. Then, if necessary, a new state is created to split the last encountered edge, state that is the locus of $head_i$. The automaton B of Figure 4, displays the creation of state 1 during the insertion of suf_1=abbabbc. Note that, if an already existing state matches the strict class of factor of $head_i$, the last

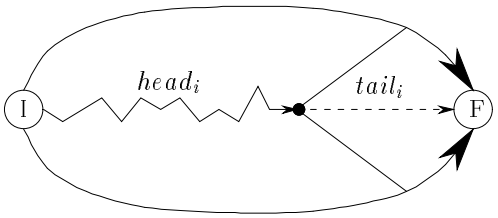

Fig. 5. Scheme of the insertion of a suf_i in $\mathcal{A}_{i-1}$.

encountered edge is split in the same way, but it is redirected to this state. Such an example appears in the same example (case D): the insertion of suf_5=`bbc` induces the redirection of the edge (2,`babbc`,F) that becomes (2,`b`,3). Then, an edge labeled by $tail_i$ is created from the locus of $head_i$ to F. We can write the preliminary algorithm as follows:

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;

Note first that *SlowFind* returns the last encountered state. This keeps accessible the transition $(q, \gamma]$ that can be split if this state is not an exact locus.

Second, as in the DAWG construction, if a non-solid edge is encountered during *SlowFind*, its target state has to be duplicated in a clone and the non-solid edge is redirected to this clone. But, if the clone has just been created at the previous step, the edge is redirected to this state. Note that, in the two cases, the redirected transition becomes solid.

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. 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, which directly induces $s_x(\mathrm{F})$=I. Note that $s_x(\mathrm{I})$ 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.

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, and when a state is created after the execution of *SlowFind*.

In the 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 $s_x(q)$=w.

Moreover, the duplication has the same properties as in the DAWG construction. Let (p, γ, q) be the transition redirected during the duplication of q. We can redirect all non-solid edges that end the partial path γ and that start from a state of the suffix path of p. This is done until the first edge that is solid. We are helped in this operation by the function *FastFind*, similar to the one used in McCreight's algorithm [13], that goes through transitions just comparing the first letters of their labels. This function returns the last encountered state and edge. Note that it is not necessary to find each time the partial path γ from a suffix of p, we just need to take the suffix link of the last encountered state and the label of the previous redirected transition.

Let ϑ be the representative of a state of the suffix path of p. Observe that the corresponding redirection is equivalent to insert $suf_{i+|\alpha|-|\vartheta|}$. Indeed, all operations done after this redirection will be the same as for the insertion of suf_i, since they go through the same path.

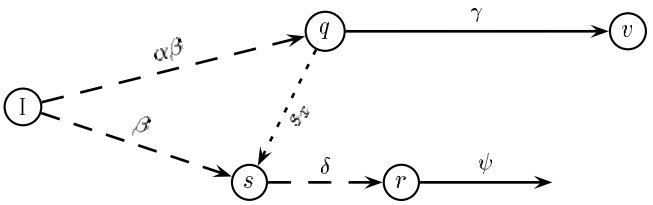

Fig. 6. Scheme of the search using suffix links

After the execution of *SlowFind*, if state v is created, we have to compute its suffix link. Let γ be the label of the transition starting at q and ending at v. To compute the suffix link, 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 6 displays a scheme of this search. The thick dashed edges represent paths in the automaton, and the thin dashed edge represents the suffix link of q. This search will allow to insert, as for the duplication, the suffixes suf_j, for $i < j < i+|head_i|$. To travel along the path, we use again the function *FastFind*. Let r and $(r, \psi]$ be the last state and transition encountered 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 we have to create a new node w. The edge $(r, \psi]$ is split, it becomes (r, ψ, w), and we insert the transition $(w, tail_i, \mathrm{F})$. Else, $(r, \psi]$ is non-solid. Then, it is split and becomes (r, ψ, v). 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 $\psi = \varepsilon$.

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.

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;
```

5.2 Complexity

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

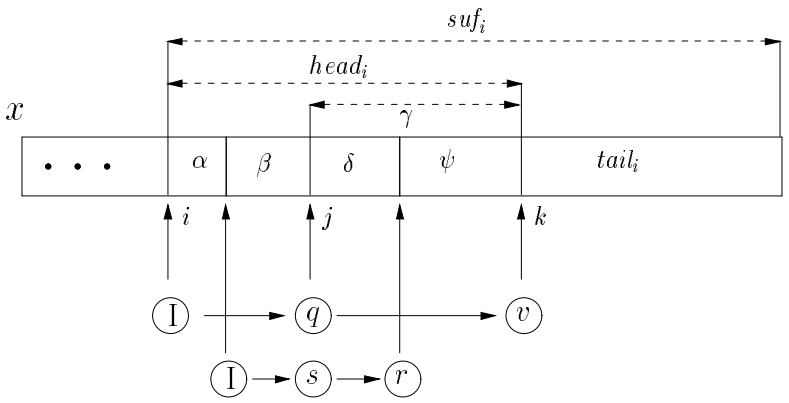

Fig. 7. Positions of labels when suf_i is inserted

Sketch of the proof
It can be proved that each step of the algorithm leads to increase strictly variables j or k in the generic situation displayed in Figure 7. These variables respectively represent the index of the current suffix being inserted, and a pointer on the text. These variables never decrease. Therefore, the total running time of the algorithm is linear in the length of x.

6 Conclusion

We have considered the Compact Direct Acyclic Word Graph, which is an efficient compact data structure to represent all suffixes of a word. There are many data structures representing this set. But, this one allows an interesting space gain compared to the well-known DAWG, which is a reference. Indeed, on the one hand, the upper bounds are of $|x| + 1$ states and $2|x| - 2$ transitions. This saves $|x|$ states and $|x|$ transitions of the DAWG, which leads to faster utilisation. On the other hand, experiments on genomic DNA sequences and random strings display a memory space gain of 50% according 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. Of course, it is easier to compute, by reduction, the CDAWG from the DAWG. On the contrary, our algorithm saves time and space simultaneously.

References

1. A. Anderson and S. Nilsson. Efficient implementation of suffix trees. *Software, Practice and Experience*, 25(2):129–141, Feb. 1995.
2. A. Apostolico. The myriad virtues of subword trees. In A. Apostolico & Z. Galil, editor, *Combinatorial Algorithms on Words.*, pages 85–95. Springer-Verlag, 1985.
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. Transducers and repetitions. *Theor. Comp. Sci.*, 45:63–86, 1986.
8. M. Crochemore and W. Rytter. *Text Algorithms*, chapter 5-6, pages 73–130. Oxford University Press, New York, 1994.
9. R. W. Irving. Suffix binary search trees. *Technical report TR-1995-7, Computing Science Department, University of Glasgow*, April 1995.
10. J. Karkkainen. Suffix cactus : a cross between suffix tree and suffix array. *CPM*, 937:191–204, July 1995.
11. 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.
12. U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, Oct. 1993.
13. E. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, Apr. 1976.
14. E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14:249–260, 1995.

This article was processed using the LaTeX macro package with LLNCS style