A Trie-Based Approach for Compacting Automata

Maxime Crochemore*, Chiara Epifanio**, Roberto Grossi***, and Filippo Mignosi†

Abstract. We describe a new technique for reducing the number of nodes and symbols in automata based on tries. The technique stems from some results on anti-dictionaries for data compression and does not need to retain the input string, differently from other methods based on compact automata. The net effect is that of obtaining a lighter automaton than the directed acyclic word graph (DAWG) of Blumer et al., as it uses less nodes, still with arcs labeled by single characters.

Keywords: Automata and formal languages, suffix tree, factor and suffix automata, index, text compression.

1 Introduction

One of the seminal results in pattern matching is that the size of the minimal automaton accepting the suffixes of a word (DAWG) is linear [4]. This result is surprising as the maximal number of subwords that may occur in a word is quadratic according to the length of the word. Suffix trees are linear too, but they represent strings by pointers to the text, while DAWGs work without the need of accessing it.

DAWGs can be built in linear time. This result has stimulated further work. For example, [8] gives a compact version of the DAWG and a direct algorithm to construct it. In [13] and [14] it is given an algorithm for online construction of DAWGs. In [11] and [12] space-efficient implementations of compact DAWGs are designed. For comparisons and results on this subject, see also [5].

In this paper we present a new compaction technique for shrinking automata based on antifactorial tries of words. In particular, we show how to apply our technique to factor automata and DAWGs by compacting their spanning tree obtained by a breadth-first search. The average number of nodes of the structure

* Institut Gaspard-Monge, Université de Marne-la-Vallée, France and King's College (London), Great Britain (`mac@univ-mlv.fr`).

** Dipartimento di Matematica e Applicazioni, Università di Palermo, Italy (`epifanio@math.unipa.it`).

*** Dipartimento di Informatica, Università di Pisa, Italy (`grossi@di.unipi.it`).

† Dipartimento di Matematica e Applicazioni, Università di Palermo, Italy (`mignosi@math.unipa.it`).

thus obtained can be sublinear in the number of symbols of the text, for highly compressible sources. This property seems new to us and it is reinforced by the fact the number of nodes for our automata is always smaller than that for DAWGs.

We build up our finding on "self compressing" tries of antifactorial binary sets of words. They were introduced in [7] for compressing binary strings with antidictionaries, with the aim of representing in a compact way antidictionaries to be sent to the decoder of a static compression scheme. We present an improvement scheme for this algorithm that extends its functionalities to any chosen alphabet for the antifactorial sets of words M. We employ it to represent compactly the automaton (or better the *trim*) $\mathcal{A}(M)$ defined in [6] for recognizing the language of all the words avoiding elements of M (we recall that a word w avoids $x \in M$ if x does not appear in w as a factor).

Our scheme is general enough for being applied to any index structure having a failure function. One such example is that of (generalized) suffix tries, which are the uncompacted version of well-known suffix trees. Unfortunately, their number of nodes is $O(n^2)$ and this is why researchers prefer to use the $O(n)$-node suffix tree. We obtain compact suffix tries with our scheme that have a linear number of nodes but are different from suffix trees. Although a compact suffix trie has a bit more nodes than the corresponding suffix tree, all of its arcs are labeled by single symbols rather than factors (substrings). Because of this we can completely drop the text, as searching does not need to access the text contrarily to what is required for the suffix tree. We exploit suffix links for this kind of searching. As a result, we obtain a family of automata that can be seen as an alternative to suffix trees and DAWGs.

This paper is organized as follows. Section 2 contains our generalization of some of the algorithms in [7] so as to make them work with any alphabet. Section 3 presents our data structure, the compact suffix trie and its connection to automata. Section 4 contains our new searching algorithms for detecting a pattern in the compact tries and related automata. Finally, we present some open problems and further work on this subject in Section 5.

2 Compressing with Antidictionaries and Compact Tries

In this section we describe a non-trivial generalization of some of the algorithms in [7] to any alphabet A, in particular with Encoder and Decoder algorithms described next. We recall that if w is a word over a finite alphabet A, the set of its factors is called $F(w)$. For instance, if $w = \mathtt{aeddebc}$, then $F(w) = \{\varepsilon, \mathtt{a}, \mathtt{b}, \ldots, \mathtt{aeddebc}\}$.

Let us take some words in the complement of $F(w)$, *i.e.*, let us take some words that are not factors of w, call these *forbidden*. This set of such words AD is called an *antidictionary* for the language $F(w)$. Antidictionaries can be finite as well as infinite. For instance, if $w = \mathtt{aeddebc}$ the words $\mathtt{aa}$, $\mathtt{ddd}$, and $\mathtt{ded}$ are forbidden and the set $\{\mathtt{aa}, \mathtt{ddd}, \mathtt{ded}\}$ is an antidictionary for $F(w)$. If

$w_1 = 001001001001$, the infinite set of all words that have two 1's in the i-th and $i+2$-th positions, for some integer i, is an antidictionary for w_1.

We want to stress that an antidictionary can be any subset of the complement of $F(w)$. Therefore an antidictionary can be defined by any property concerning words.

The compression algorithm in [7] treats the input word in an on-line manner. Let us suppose to have just read the word v, proper prefix of w. If there exists any word $u = u'a$, where $a \in \{\mathtt{0}, \mathtt{1}\}$, in the antidictionary AD such that u' is a suffix of v, then surely the letter following v cannot be a, *i.e.*, the next letter is b, with $b \neq a$. In other words, we know in advance the next letter b that turns out to be "redundant" or predictable. As remarked in [7], this argument works only in the case of binary alphabets.

We show how to generalize the above argument to any alphabet A, i.e., any cardinality of A. The main idea is that of eliminating redundant letters with the compression algorithm ENCODER. In what follows the word to be compressed is noted $w = a_1 \cdots a_n$ and its compressed version is denoted by $\gamma(w)$.

ENCODER (antidictionary AD, word $w \in A^*$)
$v \leftarrow \varepsilon$; $\gamma \leftarrow \varepsilon$;
for $a \leftarrow$ first to last letter of w
  if there exists a letter $b \in A, b \neq a$ such that for every suffix u' of v, $u'b \notin AD$ **then**
    $\gamma \leftarrow \gamma.a$;
  $v \leftarrow v.a$;
return $(|v|, \gamma)$;

As an example, let us run this algorithm on the string $w = \mathtt{aeddebc}$, with $AD = \{\mathtt{aa}, \mathtt{ab}, \mathtt{ac}, \mathtt{ad}, \mathtt{aeb}, \mathtt{ba}, \mathtt{bb}, \mathtt{bd}, \mathtt{be}, \mathtt{da}, \mathtt{db}, \mathtt{dc}, \mathtt{ddd}, \mathtt{ea}, \mathtt{ec}, \mathtt{ede}, \mathtt{ee}\}$.

The steps of the execution are described in the next array by the current values of the prefix $v_i = a_1 \cdots a_i$ of w that has been just considered and of the output $\gamma(v_i)$. In the case of a positive answer to the query to the antidictionary AD, the array indicates the value of the corresponding forbidden word u, too. The number of times the answer is positive in a run corresponds to the number of bits erased.

ε	$\gamma(\varepsilon) = \varepsilon$	
$v_1 = \mathtt{a}$	$\gamma(v_1) = \mathtt{a}$	
$v_2 = \mathtt{ae}$	$\gamma(v_2) = \mathtt{a}$	$\mathtt{aa}, \mathtt{ab}, \mathtt{ac}, \mathtt{ad} \in AD$
$v_3 = \mathtt{aed}$	$\gamma(v_3) = \mathtt{a}$	$\mathtt{ea}, \mathtt{ec}, \mathtt{ee}, \mathtt{aeb} \in AD$
$v_4 = \mathtt{aedd}$	$\gamma(v_4) = \mathtt{a}$	$\mathtt{da}, \mathtt{db}, \mathtt{dc}, \mathtt{ede} \in AD$
$v_5 = \mathtt{aedde}$	$\gamma(v_5) = \mathtt{a}$	$\mathtt{da}, \mathtt{db}, \mathtt{dc}, \mathtt{ddd} \in AD$
$v_6 = \mathtt{aeddeb}$	$\gamma(v_6) = \mathtt{ab}$	
$v_7 = \mathtt{aeddebc}$	$\gamma(v_7) = \mathtt{ab}$	$\mathtt{ba}, \mathtt{bb}, \mathtt{bd}, \mathtt{be} \in AD$

Remark that γ is not injective. For instance, $\gamma(\mathtt{aed}) = \gamma(\mathtt{ae}) = \mathtt{a}$.

In order to have an injective mapping we consider the function $\gamma'(w) = (|w|, \gamma(w))$. In this case we can reconstruct the original word w from both $\gamma'(w)$ and the antidictionary.

Remark 1. Instead of adding the length $|w|$ of the whole word w other choices are possible, such as to add the length $|w'|$ of the last encoded fragment w' of w. In the special case in which the last letter in w is not erased, we have that $|w'| = 0$ and it is not necessary to code this length. We will examine this case while examining the algorithm Decompact.

The decoding algorithm works as follows. The compressed word is $\gamma(w) = b_1 \cdots b_h$ and the length of w is n. The algorithm recovers the word w by predicting the letter following the current prefix v of w already decompressed. If there exists a unique letter a in the alphabet A such that for any suffix u' of v, the concatenation $u'a$ does not belong to the antidictionary, then the output letter is a. Otherwise we have to read the next letter from the input γ.

Decoder (antidictionary AD, word $\gamma \in A^*$, integer n)
$v \leftarrow \varepsilon$;
while $|v| < n$
  if there exists a unique letter $a \in A$ such that for any u' suffix of v $u'a$ does not belong to AD **then**
    $v \leftarrow v \cdot a$;
  else
    $b \leftarrow$ next letter of γ;
    $v \leftarrow v \cdot b$;
return (v);

The antidictionary AD must be structured in order to answer, for a given word v, whether there exist $|A| - 1$ words $u = u'b$ in AD, with $b \in A$ and $b \neq a$, such that u' is a suffix of v. In case of a positive answer the output should also include the letter a.

Languages avoiding finite sets of words are called *local* and automata recognizing them are ubiquitously present in Computer Science (cf [2]).

Given an antidictionary AD, the algorithm in [6], called L-automaton, takes as input the trie $\mathcal{T}$ that represents AD, and gives as output an automaton recognizing the language $L(AD)$ of all words avoiding the antidictionary. This automaton has the same states as those in trie $\mathcal{T}$ and the set of labeled edges of this automaton includes properly the one of the trie. The transition function of automaton $\mathcal{A}(AD)$ is called δ. This automaton is complete, i.e., for any letter a and for any state v, the value of $\delta(v, a)$ is defined.

If AD is the set of the minimal forbidden words of a text t, then it is proved in [6] that the trimmed version of automaton $\mathcal{A}(AD)$ is the *factor automaton* of t. If the last letter in t is a letter \$ that does not appear elsewhere in the text, the factor automaton coincides with the DAWG, apart from the set of final states. In fact, while in the factor automaton every state is final, in the DAWG the only final state is the last one in every topological order. Therefore, if we have a technique for shrinking automata of the form $\mathcal{A}(AD)$, for some antidictionary (AD), this technique will automatically hold for DAWG, by appending at the end of the text a symbol \$ that does not appear elsewhere. Actually the trie that we compact is the spanning tree obtained by a breadth-first search of the

DAWG, as this spanning tree is the trie obtained by pruning all leaves from the trie of all minimal forbidden words of text t (cf. [6]). The appendix presents an example of this procedure.

Self-compressing tries. Let us analyze now a technique to achieve a better compression ratio than the one obtained with the simple application of algorithm ENCODER.

If AD is an antifactorial antidictionary for a text t then for any word $v \in AD$ the set $AD \setminus \{v\}$ is an antidictionary for v. This last property lets us to compress v using $AD \setminus \{v\}$ or a subset of it. Hence the new technique, that is analogous to the one in [7], will consist in *self-compressing* the antidictionary of the text t that is not necessarily binary.

Let us go in details and consider a generic word v in AD. Words u in $AD \setminus \{v\}$ of length smaller than or equal to v's one will be used to compress v and to obtain its compressed version $\gamma_1(v)$. If a word u in AD, such that $|u| = |v|$, is used for compressing v, then u and v have the same prefix x of length $|u|-1$ of v. Further $\gamma_1(v) = \gamma_1(u) = \gamma_1(x)$.

In this case we loose information on last letters of u and v. We have two choices. The first one is to compress a word v in AD by using as an antidictionary the subset of AD including all words having length strictly smaller than $|v|$. The second choice consists in adopting some strategy to let only one of the two words being compressed using the other one. We will proceed along the first choice, i.e. while compressing a given word in the antidictionary we would not use words of its same length.

Definition 1. *A r-uple of words $(v_1, \ldots, v_r)$, being r the cardinality of the alphabet A, is called stopping r-uple if $v_1 = u_1 a_1, \ldots v_r = u_r a_r$, where a_i are distinct symbols in A and u_i is a suffix of u_{i+1}, for any i such that $1 \le i \le r-1$.*

Proposition 1. *Let t be a text and AD be an antifactorial antidictionary of t. If there exists a stopping r-uple $(v_1, \ldots, v_r)$, with $v_i = u_i a_i, a_i \in A, 1 \le i \le r$, then u_r is a suffix of t and does not appear elsewhere in t. Moreover there exists at most one such r-uple of words.*

Proposition 2. *Let $(v_1, \ldots, v_r)$ be the only possible stopping r-uple in the antidictionary AD of a text t. If the prefix u_{r-1} of length $|v_{r-1}|-1$ of the penultimate word v_{r-1} in the lexicographic word is a proper suffix of the prefix u_r of length $|v_r|-1$ of the last word v_r, then last letter in v_r is never erased.*

Let us now examine another algorithm, similar to that presented in [7], apart from some details, that is very important in our case. The idea of this algorithm is to "erase" states with only one outgoing edge whose label is predictable by the previous compression algorithm that uses as antidictionary the leaves of the trie having strictly smaller depth. In other words, consider a state p that has a unique outgoing edge labeled by a such that, for any other symbol $b \neq a$, the longest suffix of pb stored in the trie (i.e., $\delta(p, b)$) is a leaf. Then we erase state p together with its outgoing edge and we replace p by its unique child as described in Self-compress. This algorithm works in the case when in the antidictionary there are no stopping r-uples.

```
Self-compress (trie T, function δ)
    i ← root of T;
    create root i';
    add (i, i') to empty queue Q;
    while Q ≠ ∅
        extract (p, p') from Q;
        if p has more than one child then
            for each child q_j of p
                create q'_j as child of p';
                add (q_j, q'_j) to Q;
        else if q is a unique child of p and
            q = δ(p, a), a ∈ A then
                if δ(p, b) is a leaf ∀ b ∈ A, b ≠ a then
                    add + sign to q';
                    add (q, p') to Q;
                else create q' as a-child of p';
                     add (q, q') to Q;
    return trie having root i';
```

In this case we do not delete the child, but the parent of a compressed edge, and we add a + sign to the remaining node. Note that with this new algorithm there is no guarantee that function δ is always defined on the nodes belonging to the compacted trie (this property was guaranteed with previous Self-compress algorithm). To avoid this drawback, δ can be recovered by the relation $s(v)$ (usually called suffix link), the longest suffix of v that is a node in the trie, where the left-hand δ is the new version defined in terms of the old version, the right-hand δ,

$$\delta(p, a) = \begin{cases} \delta(p, a) & \text{if it is defined} \\ \\ \delta(s^n(p), a) & \text{if not, where } n = \min\{m \mid \delta(s^m(p), a) \text{ is defined}\} \end{cases}$$

Hence, the output of this algorithm represents a compact version of the input trie and, if we include the new function δ, the compact version of the automaton $\mathcal{A}(AD)$.

To deal with the case when the antidictionary contains a stopping r-uple, we have to add some more information to the output of the algorithm. This added information can be to keep a pointer to the leaf corresponding to the longest element of the stopping r-uple together with the length of the compressed ingoing edge to this leaf. Details are left to the reader.

Compact tries present some very useful and important properties. In particular we are able to check whether a pattern p is recognized by the automaton $\mathcal{A}(AD)$, by only “decompacting” a small part of the compact trie, output of previous algorithm, and not by “decompacting” the whole trie described in [7].

The algorithm Search that realizes this task is described in Section 4. Before that, we introduce compact suffix tries in Section 3, because we make use of their properties in the next section.

Recall that if the text ends with a symbol \$ the compacted version of automaton $\mathcal{A}(AD)$ can be seen as a compacted version of the DAWG, that is different from the classical one, called CDAWG.

Proposition 3. *Every node v that is in our compacted DAWG and is not in the CDAWG is such that it has only one outgoing edge and its suffix link points to a node of the CDAWG.*

3 Compact Suffix Tries

We want to extend our technique to suffix tries that do not represent antifactorial sets of words.

We adopt the notation defined in [15], and assume that the reader is familiar with suffix trees. Given a word $w = a_1 a_2 \ldots a_n$ of n symbols drawn from an alphabet A, called letters, we denote by $w[j \ldots j + k - 1]$ the factor of w of length k that appears in w at position j. The length of w is denoted by $|w| = n$. The empty word ϵ is a factor of any word. A factor of the form $w[j, |w|]$ (resp. $w[1, j]$) is called a suffix (resp. prefix) of w.

The suffix trie $ST(w)$ of a word w is a trie where the set of leaves is the set of suffixes of w that do not appear previously as factors in w.

We can identify a node of the trie with the label of the unique path from the root of the trie to the given node. Sometimes we can skip this identification and, given a node v, we call σ_v the label word that represents the unique path from the root to v.

In this section we define our compacted version $CST(w)$ of the suffix trie. Basically we use the same approach of the suffix tree, but we compact a bit less, i.e., we keep all nodes of the suffix tree $S(w)$ and we keep some more nodes of the trie. In this way we can avoid to keep the text aside.

In order to describe this new data structure, we have to define first of all its nodes, namely,

1. all the nodes of suffix tree $S(w)$, and
2. all the nodes v of trie $ST(w)$ such that $s(v)$ is a node of suffix tree $S(w)$.

Recall that in the suffix tree, for any suffix $w[j, |w|]$ of w there is a node v such that $\sigma_v = w[j, |w|]$, even if $w[j, |w|]$ has appeared as factor in another position.

At this point one may wonder why are we keeping the nodes of type 2. What is the relation between these nodes and minimal forbidden words? The answer to the question is given by next proposition, that can be seen as a kind of converse of Proposition 3.

Proposition 4. *For any node v described in point 2, there exists a letter a such that $\sigma_v a$ is a minimal forbidden word of w.*

A suffix trie is not a trie of minimal forbidden words, but if we add to the trie the minimal forbidden words, we can describe a compacting algorithm that is analogous to that presented in Section 2. The nodes that are neither of type 1, nor of type 2, and that are not leaves can be erased. More precisely, we do not need to introduce explicitly the minimal forbidden words in the trie.

Let us now define the arcs of $CST(w)$. Since the nodes of $CST(w)$ are all nodes of the trie $ST(w)$, for any node v we assign the same number of outgoing arcs, each of them labeled by a different letter. Hence, for any arc of the form (v, v') with label $l((v, v')) = a$ in $ST(w)$, we have an arc (v, x) with label $l((v, x)) = a$ in $CST(w)$, where the node x is

(i) v' itself, in the case it is still present in $CST(w)$;
(ii) the first node in $CST(w)$ that is a descendant of v' in $ST(w)$, when v' is not a node of $CST(w)$.

In case (ii), we consider as this arc (v, x) represents the whole path from v to x in the trie $\mathcal{T}r(w)$, and we say that this arc has length greater than one. We further add a $+$ sign to node x, in order to record this information.

To complete the definition of $CST(w)$ we keep the suffix link function over these nodes. Notice that, by definition, for any node v of $CST(w)$, the node pointed by a suffix link, $s(v)$, is always a node of the suffix tree $S(w)$ and hence it also belongs to $CST(w)$.

We now show that the number of nodes of $CST(w)$ is still linear in the length $|w|$ of w, independently from the alphabet size. Indeed, since the number of nodes of the suffix tree $S(w)$ is linear, we only have to prove that the number of nodes that we add to it to obtain the nodes of $CST(w)$ is linear in the length $|w|$ of w. This is what we prove in the lemma.

Lemma 1. *The number of nodes v of the trie $ST(w)$ such that v does not belong to the tree $S(w)$ and $s(v)$ is a node of the tree $S(w)$ is smaller than the text size.*

Notice that Lemma 1 states a result that is "alphabet independent", i.e., the result does not depend on the alphabet size. By definition, since the number of nodes in the tree is smaller than $2|w|$, it is straightforward to prove that the number of added nodes is smaller than this quantity multiplied by the cardinality of the alphabet.

The power of our new data structure is that it still has some nice features when compared to the suffix tree. In fact, we possibly reduce the amount of memory required by arc labels. While suffix trees arcs are labeled by substrings, compact suffix tries ones are labeled by single symbols. So they can be accessed "locally" without jumping to positions in the text, which is no more needed. Also, this property is "alphabet independent", as stated in next lemma.

Lemma 2. *Let A be any alphabet and w be a word on it. The number of nodes of the compact suffix trie $CST(w)$ is linear in the length n of w, whichever cardinality A has.*

We remark here that, if we add terminal states to the DAWG of a word w, the structure thus obtained is the minimal automaton accepting the suffixes of w. This implies that if we reduce the size of this automaton, we consequently reduce that of the DAWG.

4 Searching in Compact Tries and Related Automata

In this section we show how to check whether a pattern p occurs as a factor in a text w, by just "decompacting" a small part of the compact suffix trie $CST(w)$ associated with w (not the whole trie). Note that the arcs in $CST(w)$ are labeled with single letters instead of substrings, so we must recover them somehow without accessing the text.

Generally speaking, with the same technique we can check whether a pattern p is recognized by an automaton $\mathcal{A}(AD)$. In this section we center our description around the algorithms for compact suffix tries, but they work on automata with minor modifications.

Let us begin by examining a function that returns the string associated with the path in the trie between two nodes u and v.

```
DECOMPACT (Compacted trie CST, function δ, function s, arc (u, v))
w ← ϵ;
a ← label of (u, v);
if v has not a + then
      w ← a;
else
      z ← s(u);
      while z ≠ s(v)
            w ← w concat DECOMPACT(CST, δ, s,(z, δ(z, a)));
            z ← δ(z, a);
            if z has only one child x in CST then
                  a ← label of (z, x);
return w
```

Proposition 5. *Let u be a node in the compacted trie. If v is a child of u, then the path from $s(u)$ to $s(v)$ in the compacted trie has no forks.*

An easy variation of this algorithm can be used to check whether a pattern occurs in a text. More precisely, algorithm SEARCH takes as input a node u and a pattern p. If there is no outgoing path in the decompacted trie from u labeled p, it returns "null." If such a path exists and v is its incoming node, it returns v. The pattern p is represented either as a static variable or as a pointer to pointers. We denote by $succ(p)$ the operation that, received as input a pointer to a letter, points to next letter and $p[0]$ points to the first letter of p. The proof of the correctness of this algorithm follows by induction and by Proposition 5.

```
Search (Compacted trie CST, function δ, function s, node u, pattern p)
if p[0] = end of string then
    return(u)
else
    if ∄v such that p[0] is label of (u, v) then
        return (null)
    else
        let v such that (u, v) has label p[0];
        if v has not a + then
            Search(CST, δ, s, v, succ(p))
        else
            z ← s(u);
            while z ≠ s(v)
                z′ ← δ(z, p[0]);
                Search(CST, δ, s,z,p)
                z ← z′
```

Notice that, since Search returns a pointer, we can use this information to detect all the occurrences in an index. The drawback of Decompact and Search is that they may require more then $O(m)$ time to decompact m letters and this does not guarantee a searching cost that is linear in the pattern length. To circumvent this drawback, we introduce a new definition, that we use for a linear-time version of Decompact (hence, Search).

Definition 2. *Let s be the suffix link function of a given trie. We define* **super-suffix link** *for a given node v in the trie, the function*

$$s^* : v \longrightarrow s^k(u),$$

where u is a node in the trie and k is the greatest integer such that $s^k(u)$ and $s^k(v)$ are connected by a single edge (i.e., of path length 1), while the path from $s^{k+1}(u)$ to $s^{k+1}(v)$ has a length strictly greater than 1.

In order to appreciate how this new definition is useful for our goal, consider the task of decompacting the edge (u, v) in CST to reconstruct the substring originally associated with the uncompacted tries. We do this by considering repeatedly $(s(u), s(v))$, $(s^2(u), s^2(v))$, and so on, producing symbols only when the path from the two nodes at hand has at least one more edge. Unfortunately, we cannot guarantee that $(s(u), s(v))$ satisfies this property. With super-suffix links, we know that this surely happens with $(s(s^*(u)), s^*(s(v)))$ by Definition 2. Now, Decompact works in linear time with the aid of super-suffix links.

```
SuperDecompact (Compacted trie CST, function δ, function s, arc (u,v))
w ← ε;
a ← label of (u,v);
if v has not a + then
    w ← a;
else
    z ← s(s*(v));
    while z ≠ s(δ(s*(v),a))
        w ← w concat SuperDecompact(CST, δ, s,(z,δ(z,a)));
        z ← δ(z,a);
        if z has only one child x in CST then
            a ← label of (z,x);
return w
```

As a result of the linearity of SuperDecompact, we also have:

Lemma 3. *Algorithm* Search *for a pattern p takes $O(|p|)$ time by using super-suffix links instead of suffix links.*

Full-Text Indexing. A full-text index over a fixed text t is an abstract data type based on the set of all factors of t, denoted by $Fact(t)$. Such data type is equipped with some operations that allow it to answer the following query: given $x \in A^*$, find the list of all occurrences of x in t. If the list is empty, then x is not a factor of t.

Suffix trees and compacted DAWGs (CDAWGs) can answer to previous query in time proportional to the length of x plus the size of the list of the occurrences. By superposing the structure of CDAWGs to our compacted DAWGs we can obtain the same performance. More precisely, for any edge q of our compacted DAWG, let us define recursively $final(q)$

$$final(q) = \begin{cases} q & \text{if } q \text{ has more than two children or } q \text{ is the only final state} \\ final(\delta(q,a)) & \text{if not, and } (\delta(q,a)) \text{ is the only outgoing edge from } q. \end{cases}$$

For any edge (p,q) we add another edge $p, final(q)$ labelled by the length of the path from p to $final(q)$ in the DAWG. These new edges allow to simulate CDAWGs.

We have examples of infinite sequences of words where our compacted DAWGs have size exponentially smaller than the text size. This is the first time to our knowledge that full-text indexes show this property.

5 Conclusions and Further Work

In this paper we have presented a new technique for compacting tries and their corresponding automata. They have arcs labeled with single characters; they do not need to retain the input string; also, they have less nodes than similar automata. We are currently performing some experiments for testing the effective gain in terms of final size of the automata. For highly compressible sources, our automata seem to be sublinear in space. We do not known if algorithm Search strictly needs super-suffix links to work in linear time on the average (in the worst case, they are needed). Finally, we are investigating methods for a direct construction of our automata.

References

1. A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search, *Comm. ACM* **18:6** (1975) 333–340.
2. M.-P. Béal. Codage Symbolique, Masson, 1993.
3. M.-P. Béal, F. Mignosi, and A. Restivo. Minimal Forbidden Words and Symbolic Dynamics, in (*STACS'96*, C. Puech and R. Reischuk, eds., LNCS **1046**, Springer, 1996) 555–566.
4. A. Blumer, J. Blumer, D. Haussler, A. Ehrenfeucht, M. T. Chen, and J. Seiferas. The Smallest Automaton Recognizing the Subwords of a Text, Theoretical Computer Science, **40**, 1, 1985, 31–55.
5. M. Crochemore, Reducing space for index implementation, in Theoretical Computer Science, **292**, 1, 2003, 185–197.
6. M. Crochemore, F. Mignosi, A. Restivo, Automata and forbidden words, in Information Processing Letters, **67**, 3, 1998, 111–117.
7. M. Crochemore, F. Mignosi, A. Restivo, S. Salemi, Data compression using antidictonaries, in Special issue *Lossless data compression*, J. Storer ed., Proceedings of the IEEE, **88**, 11, 2000), 1756–1768.
8. M. Crochemore, R. Vérin, Direct Construction of Compact Directed Acyclic Word Graphs, in CPM97, A. Apostolico and J. Hein, eds., LNCS **1264**, Springer-Verlag, 1997, 116–129.
9. V. Diekert, Y. Kobayashi. *Some identities related to automata, determinants, and Möbius functions*, Report 1997/05, Fakultät Informatik, Universität Stuttgart, 1997, in (*STACS'96*, C. Puech and R. Reischuk, eds., LNCS **1046**, Springer-Verlag, 1996) 555–566.
10. B. K. Durgan. *Compact searchable static binary trees*, in Information Processing Letters, **89**, 2004, 49–52.
11. J. Holub. *Personal Communication*, 1999.
12. J. Holub, M. Crochemore. *On the Implementation of Compact DAWG's*, in Proceedings of the *7th Conference on Implementation and Application of Automata*, University of Tours, Tours, France, July 2002, LNCS **2608**, Springer-Verlag, 2003, 289–294.
13. S. Inenaga, H. Hoshino, A. Shinohara, M. Takeda, S. Arikawa, G. Mauri, G. Pavesi. *On-Line Construction of Compact Directed Acyclic Word Graphs*, To appear in Discrete Applied Mathematics (special issue for CPM'01).
14. S. Inenaga, H. Hoshino, A. Shinohara, M. Takeda, S. Arikawa, G. Mauri, G. Pavesi. *On-Line Construction of Compact Directed Acyclic Word Graphs*, Proceedings of *CPM 2001*, LNCS **2089**, Springer-Verlag, 2001, 169–180.
15. M. Lothaire. *Algebraic Combinatorics on Words.* Encyclopedia of Mathematics and its Applications, **90**, Cambridge University Press (2002).

Appendix

Let us examine, as an example, the text $t = abaababaaba\$$. Minimal words of text t are: aaa, $aabaa$, $aa\$$, $babaabab$, $babab$, $baba\$$, bb, $b\$$, $\$a$, $\$b$, $\$\$$.

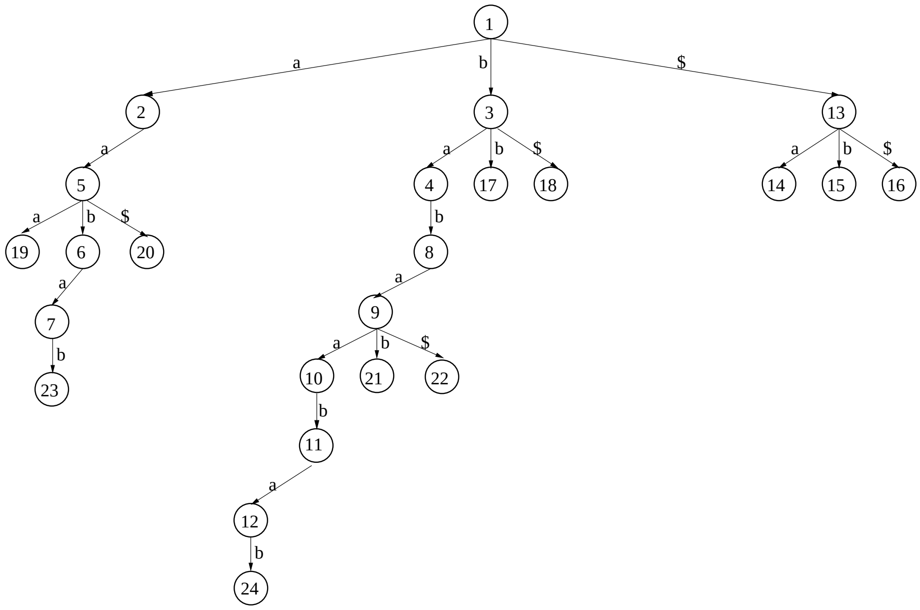

Fig. 1. Trie of the minimal forbidden factors.

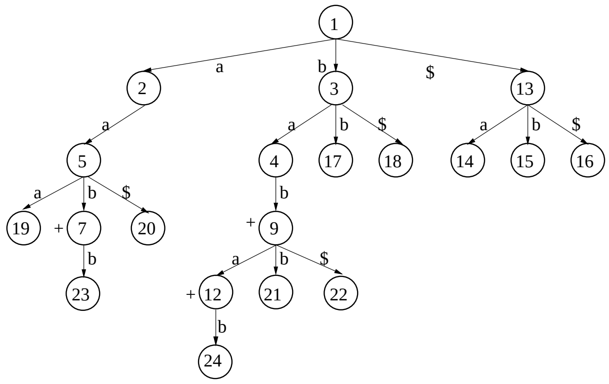

Fig. 2. Self-Compressed trie of the minimal forbidden factors.

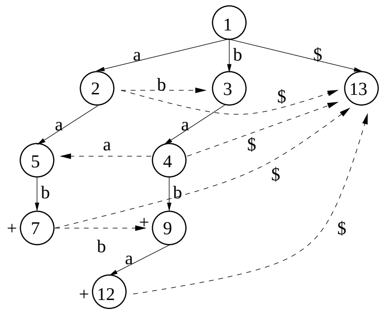

Fig. 3. Compacted Self-Compressed DAWG of the text t. It has 9 states. It has been obtained by adding the function δ whenever undefined (dotted arcs) to the self-compressed trie and by trimming it, i.e. by pruning the leaves and the corresponding ingoing and outgoing arcs. Suffix links and super-suffix links are not drawn.

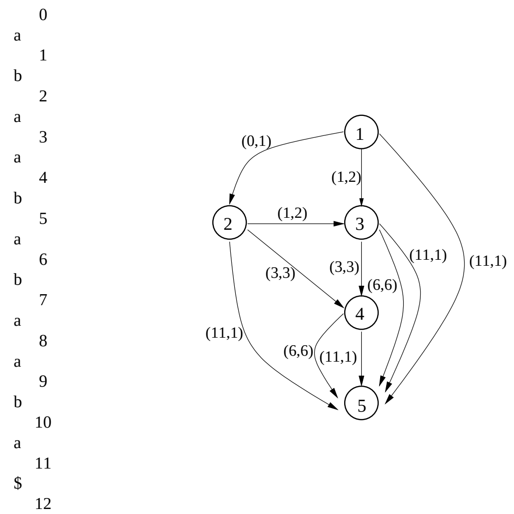

Fig. 4. CDAWG of the text t.