Data Compression Using Antidictionaries

M. Crochemore , F. Mignosi , A. Restivo , S. Salemi

Abstract— **We give a new text compression scheme based on Forbidden Words ("antidictionary"). We prove that our algorithms attain the entropy for balanced binary sources. They run in linear time. Moreover, one of the main advantages of this approach is that it produces very fast decompressors. A second advantage is a synchronization property that is helpful to search compressed data and allows parallel compression. The techniques used in this paper are from Information Theory and Finite Automata.**

Keywords— **Data Compression, Lossless compression, Information Theory, Finite Automaton, Forbidden Word, Pattern Matching.**

I. Introduction

We present a simple text compression method called DCA (Data Compression with Antidictionaries) that uses some "negative" information about the text, which is described in terms of antidictionaries. In contrast to other methods that make use, as a main tool, of dictionaries, *i.e.*, particular sets of words occurring as factors in the text (cf. [1], [2], [3], [4] and [5]), our method takes advantage of words that do not occur as factors in the text, *i.e.*, that are forbidden. Such sets of words are called here antidictionaries.

We describe a static compression scheme that runs in linear time (Sections II and III) including the construction of antidictionaries (Section V and Section VI). Variations using statistical or dynamical considerations are discussed in the conclusion (Section VII)

Let w be a text on the binary alphabet $\{0,1\}$ and let AD be an antidictionary for w. By reading the text w from left to right, if at a certain moment the current prefix v of the text has as suffix a word u' such that $u = u'a \in AD$ with $a \in \{0,1\}$, *i.e.*, u is forbidden, then surely the letter following v in the text cannot be a and, since the alphabet is binary, it is the letter $b \neq a$. In other terms, we know in advance the next letter b, that turns out to be redundant or predictable. The main idea of our method is to eliminate redundant letters in order to achieve compression. The decoding algorithm recovers the text w by predicting the letter following the current prefix v of w already decompressed.

The method proposed here presents some analogies with ideas discussed by C. Shannon at the very beginning of Information Theory. In [6] Shannon designed psychological experiments in order to evaluate the entropy of English. One of such experiments was about the human ability to reconstruct an English text where some characters were erased. Actually our compression method erases some characters and the decompression reconstruct them.

We prove (Section IV) that the compression rate of our compressor reaches the entropy almost surely, provided that the source is balanced and produced from a finite antidictionary. This type of source approximates a large class of sources, and consequently, a variant of the basic scheme gives an optimal compression for them. The idea of using antidictionaries is founded on the fact that there exists a topological invariant for Dynamical Systems based on forbidden words, invariant that is independent of the entropy (cf. [7] and [8]).

The use of the antidictionary AD in coding and decoding algorithms requires that AD must be structured in order to answer to the following query on a word v: does there exists a word $u = u'a$, $a \in \{0,1\}$, in AD such that u' is a suffix of v? In the case of positive answer the output should also include the letter b defined by $b \neq a$. One of the main features of our method is that we are able to implement efficiently finite antidictionaries in terms of finite automata. This leads to fast linear-time compression and decompression algorithms that can be realized by sequential transducers (generalized sequential machines). This is especially relevant for fixed sources. It is then comparable to the fastest compression methods because the basic operation at compression and decompression time is just table lookup.

A central notion of the present method is that of minimal forbidden words, which allows to reduce the size of antidictionaries. This notion has also some interesting combinatorial properties. Our compression method includes algorithms to compute antidictionaries, algorithms that are based on the above combinatorial properties and that are described in detail in [9] and [10].

The compression method shares also an interesting synchronization property, in the case of finite antidictionaries. It states that the encoding of a block of data does not depend on the left and right contexts except for a limited-size prefix of the encoded block. This is also helpful to search compressed data and the same property allows to design efficient parallel compression algorithms.

The paper is organized as follows.

In Section II we give the definition of Forbidden Words and of antidictionaries. We describe DCA, our text com-

DCA URL is `http://www-igm.univ-mlv.fr/~mac/DCA.html`

M. Crochemore, Institut Gaspard-Monge, Université de Marne-la-Vallée, France. E-mail: Maxime.Crochemore@univ-mlv.fr.

F. Mignosi, Università degli Studi di Palermo, Italy and Brandeis University, U.S.A. E-mail: mignosi@altair.math.unipa.it and mignosi@cs.brandeis.edu. Work partially supported by the CNR-NATO fellowship n. 215.31 and by the project "Modelli innovativi di calcolo: metodi sintattici e combinatori" MURST, Italy.

A. Restivo, Università degli Studi di Palermo, Italy. E-mail: restivo@altair.math.unipa.it. Work partially supported by the project "Modelli innovativi di calcolo: metodi sintattici e combinatori" MURST, Italy.

S. Salemi, Università degli Studi di Palermo, Italy. E-mail: salemi@altair.math.unipa.it. Work partially supported by the project "Modelli innovativi di calcolo: metodi sintattici e combinatori" MURST, Italy.

pression and decompression algorithms (binary oriented) assuming that the antidictionary is given. In Section III we describe a data structure for finite antidictionaries that allows us to answer in an efficient way the queries needed by our compression and decompression algorithms; we show how to implement it given a finite antidictionary. In the case of rational antidictionaries the compression is also described in terms of transducers. We end the section by proving the synchronization property. In Section IV we evaluate the compression rate of our compression algorithm relative to a given antidictionary. In Section V we show how to construct antidictionaries for single words and sources. As a consequence we obtain a family of linear time optimal algorithms for text compression that are universal for balanced Markov sources with finite memory. In Section VI we give linear time improved algorithms for building antidictionaries for a static approach. They use the ideas of pruning and self-compressing. We discuss improvements and generalizations in Section VII.

Some of the results present in this paper have been succinctly stated in [11].

II. Basic Algorithms

Let us first introduce the main ideas of our algorithm on its static version. We discuss variations of this first approach in Section VII.

Let w be a finite binary word and let $F(w)$ be the set of factors of w. For instance, if $w = \texttt{01001010}$ then $F(w) = \{\varepsilon, \texttt{0}, \texttt{1}, \texttt{00}, \texttt{01}, \texttt{10}, \texttt{001}, \texttt{010}, \ldots, \texttt{01001010}\}$ where ε denotes the empty word.

Let us take some words in the complement of $F(w)$, *i.e.*, let us take some words that are not factors of w and that we call *forbidden*. This set of such words AD is called an *antidictionary* for the language $F(w)$. Antidictionaries can be finite as well infinite. For instance, if $w = \texttt{01001010}$ the words 11, 000, and 10101 are forbidden and the set $\{\texttt{11}, \texttt{000}, \texttt{10101}\}$ is an antidictionary for $F(w)$. For instance, if $w_1 = 001001001001$ the infinite set of all words that have two 1's as i-th and as $i+2$-th letter for some integer i, is an antidictionary for w_1. We want here to stress that an antidictionary can be any subset of the complement of $F(w)$. Therefore an antidictionary can be defined by any property that concerns words.

The compression algorithm treats the input word in an on-line manner. At a certain step in this process we have read the word v proper prefix of w. If there exists any word $u = u'a$, $a \in \{0,1\}$, 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, $b \neq a$. In other words, we know in advance the next letter b that turns out to be "redundant" or predictable. Remark that this argument works only in the case of binary alphabets.

The main idea in the algorithm we describe is to eliminate redundant letters. In what follows we first describe the compression algorithm, Encoder, and then the decompression algorithm, Decoder. 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 $w \in \{0,1\}^*$)
$v \leftarrow \varepsilon$; $\gamma \leftarrow \varepsilon$;
for $a \leftarrow$ first to last letter of w
    if for every suffix u' of v, $u'0$,$u'1 \notin AD$
        $\gamma \leftarrow \gamma.a$;
    $v \leftarrow v.a$;
return $(|v|, \gamma)$;

As an example, let us run the algorithm Encoder on the string $w = \texttt{01001010}$ with the antidictionary $AD = \{\texttt{000}, \texttt{10101}, \texttt{11}\}$. The steps of the treatment 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 $\gamma(w)$. In the case of positive answer to the query to the antidictionary AD, the array also indicates the value of the corresponding forbidden word u. The number of times the answer is positive in a run corresponds to the number of bits erased.

ε	$\gamma(w) = \varepsilon$	
$v_1 = \texttt{0}$	$\gamma(w) = \texttt{0}$	
$v_2 = \texttt{01}$	$\gamma(w) = \texttt{01}$	$u = \texttt{11} \in AD$
$v_3 = \texttt{010}$	$\gamma(w) = \texttt{01}$	
$v_4 = \texttt{0100}$	$\gamma(w) = \texttt{010}$	$u = \texttt{000} \in AD$
$v_5 = \texttt{01001}$	$\gamma(w) = \texttt{010}$	$u = \texttt{11} \in AD$
$v_6 = \texttt{010010}$	$\gamma(w) = \texttt{010}$	
$v_7 = \texttt{0100101}$	$\gamma(w) = \texttt{0101}$	$u = \texttt{11} \in AD$
$v_8 = \texttt{01001010}$	$\gamma(w) = \texttt{0101}$	$u = \texttt{10101} \in AD$
$v_9 = \texttt{010010100}$	$\gamma(w) = \texttt{0101}$	$u = \texttt{000} \in AD$
$v_{10} = \texttt{0100101001}$	$\gamma(w) = \texttt{0101}$	$u = \texttt{11} \in AD$

Remark that the function γ is not injective.
For instance $\gamma(\texttt{01}) = \gamma(\texttt{010}) = \texttt{01}$.

In order to have an injective mapping we can 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.

The decoding algorithm works as follow. 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 one word $u = u'a$, $a \in \{0,1\}$, in the antidictionary AD such that u' is a suffix of v, then, the output letter is b, $b \neq a$. Otherwise, the next letter is read from the input γ.

Decoder (antidictionary AD, word $\gamma \in \{0,1\}^*$,
    integer n)
$v \leftarrow \varepsilon$;
while $|v| < n$
    if for some u' suffix of v and $a \in \{0,1\}$, $u'a$
    belongs to AD
        $v \leftarrow v \cdot \neg a$;
    else
        $b \leftarrow$ next letter of γ;
        $v \leftarrow v \cdot b$;
return (v);

The antidictionary AD must be structured in order to answer to the following query on a word v: does there exist one word $u = u'a$, $a \in \{0,1\}$, in *AD* such that u' is a suffix of v? In case of a positive answer the output should also include the letter b defined by $b \neq a$. Notice that the letter a considered at line 3 is unique because, at this point, the end of the text w has not been reached so far.

In this approach, where the antidictionary is static and available to both the encoder and the decoder, the encoder must send to the decoder the length of the word $|w|$, in addition to the compressed word $\gamma(w)$, in order to give to the decoder a "stop" criterion. Slight variations of the previous compression-decompression algorithm can be easily obtained by giving other "stop" criteria: For instance, the encoder can send the number of letters that the decoder has to reconstruct after that the last letter of the compressed word $\gamma(w)$ has been read. Or the encoder can let the decoder stop when there is no more letter available in γ (line 6), or when both letters are impossible to be reconstructed according to *AD*. Doing so, the encoder must send to the decoder the number of letters to erase in order to recover the original message. For such variations antidictionaries can be structured to answer slightly more complex queries.

Since we are considering here the static case, the encoder must send to the decoder the antidictionary unless the decoder has already a copy of the antidictionary or it has an algorithmic way to reconstruct the antidictionary from some previously acquired information.

The method presented here brings to mind some ideas proposed by C. Shannon at the very beginning of Information Theory. In [6] Shannon designed psychological experiments in order to evaluate the entropy of English. One of such experiments was about the human ability to reconstruct an English text where some characters were erased. Actually our compression methods erases some characters and the decompression reconstruct them. For instance in previous example the input string is $01\bar{0}0\bar{1}01\bar{0}0\bar{1}$, where bars indicate which letters are erased during the compression.

In order to get good compression rates (at least in the static approach when the antidictionary has to be sent) we need to minimize in particular the size of the antidictionary. Remark that if there exists a forbidden word $u = u'a$, $a \in \{0,1\}$ in the antidictionary such that u' is also forbidden then our algorithm will never use this word u in the algorithms. So that we can erase this word from the antidictionary without any loss for the compression of w. This argument leads to consider the notion of *minimal forbidden word* with respect to a factorial language L, and the notion of anti-factorial language, points that are discussed in the next section.

III. Implementation of Finite Antidictionaries

When the antidictionary is a finite set, the queries on the antidictionary required by the algorithms of the previous section are realized as follows. We build a deterministic automaton accepting the words having no factor in the antidictionary. Then, while reading the text to encode, if a transition leads to a sink state, the output is the other letter. We denote by $\mathcal{A}(AD)$ the automaton built from the antidictionary *AD*. An algorithm to build $\mathcal{A}(AD)$ is described in [9] and [10]. The same construction has been discovered by Choffrut *et al.* [12], it is similar to a description given by Aho and Corasick ([13], see [14]), by Diekert *et al.* [15], and it is related to a more general construction given in [16].

The required automaton accepts a factorial language L. Recall that a language L is factorial if L satisfies the following property: for any words, u, v, $uv \in L \Rightarrow u \in L$ and $v \in L$. The complement language $L^c = A^* \setminus L$ is a (two-sided) ideal of A^*. Denoting by $MF(L)$ the base of this ideal, we have $L^c = A^* MF(L) A^*$. The set $MF(L)$ is called the set of ***minimal forbidden words*** for L. A word $v \in A^*$ is forbidden for the factorial language L if $v \notin L$, which is equivalent to say that v occurs in no word of L. In addition, v is minimal if it has no proper factor that is forbidden.

One can note that the set $MF(L)$ uniquely characterizes L, just because $L = A^* \setminus A^* MF(L) A^*$. This set $MF(L)$ is an *anti-factorial language* or a *factor code*, which means that it satisfies: $\forall u, v \in MF(L)$, $u \neq v \Longrightarrow u$ is not a factor of v, property that comes from the minimality of words of $MF(L)$. Indeed, there is a duality between factorial and anti-factorial languages, because we also have the equality: $MF(L) = AL \cap LA \cap (A^* \setminus L)$. In view of the remark made at the end of the previous section, from now on in the paper we consider only antidictionaries that consist of minimal forbidden words. Thus they are anti-factorial languages.

Figure 1 displays the trie that accepts the anti-factorial language $AD = \{000, 10101, 11\}$. The automaton produced from the trie is shown in Figure 2.

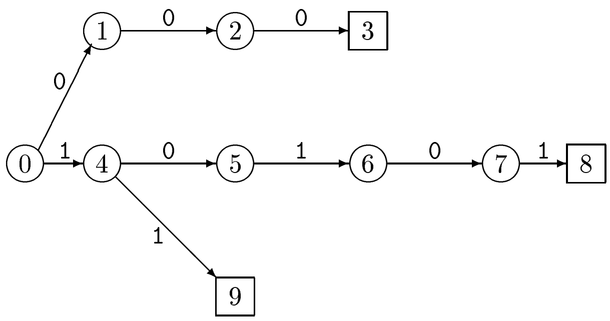

Fig. 1. Trie of the factor code $\{000, 10101, 11\}$. Squares represent terminal states.

The following theorem is proved in [10]. It is based on an algorithm called L-AUTOMATON that has as (finite) input *AD* in the form of a trie $\mathcal{T}$. It is straigthforward to get $\mathcal{T}$ if *AD* is given in the form of a list of words. The algorithm can be adapted to test whether $\mathcal{T}$ represents an anti-factorial set, to generate the trie of the anti-factorial language associated with a set of words, or even to build the automaton associated with the anti-factorial language corresponding to any set of words.

Theorem 1: The construction of $\mathcal{A}(AD)$ from $\mathcal{T}$ can be realized in linear time.

We report here, for sake of completeness, the algorithm L-AUTOMATON described in [10]. Its input, the trie $\mathcal{T}$ that represents *AD*, is a tree-like automaton accepting the set

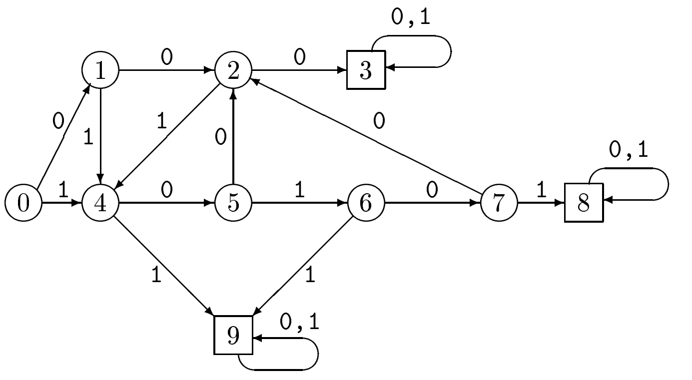

Fig. 2. Automaton accepting the words that avoid the set $\{000, 10101, 11\}$. Squares represent non-terminal states (sink states).

AD and, as such, it is noted (Q, A, i, T, δ'). The set T of terminal states is the set of leaves of the trie.

The algorithm uses a function f called a *failure function* and defined on states of $\mathcal{T}$ as follows. States of the trie $\mathcal{T}$ are identified with the prefixes of words in AD. For a state au ($a \in A$, $u \in A^*$), $f(au)$ is the longest suffix of u that is a state of the trie $\mathcal{T}$, a word that may happen to be u itself. This state is also $\delta(i, u)$, where δ is the transition function of $\mathcal{A}(AD)$, and this can be easily proved by induction on the length of u. Note that $f(i)$ is undefined, which justifies a specific treatment of the initial state in the algorithm.

```
L-AUTOMATON (trie T = (Q, A, i, T, δ'))
  for each a ∈ A
      if δ'(i, a) defined
          δ(i, a) ← δ'(i, a);
          f(δ(i, a)) ← i;
      else
          δ(i, a) ← i;
  for each state p ∈ Q \ {i} in width-first
      search and each a ∈ A
      if δ'(p, a) defined
          δ(p, a) ← δ'(p, a);
          f(δ(p, a)) ← δ(f(p), a);
      else if p ∉ T
          δ(p, a) ← δ(f(p), a);
      else
          δ(p, a) ← p;
  return (Q, A, i, Q \ T, δ);
```

A. Transducers

From the automaton $\mathcal{A}(AD)$ we can easily construct a (finite-state) transducer $\mathcal{B}(AD)$ that realizes the compression algorithm ENCODER, *i.e.*, that computes the function γ. The input part of $\mathcal{B}(AD)$ coincides with $\mathcal{A}(AD)$, with sink states removed, and the output is given as follows: if a state of $\mathcal{A}(AD)$ has two outgoing edges, then the output labels of these edges coincide with their input label; if a state of $\mathcal{A}(AD)$ has only one outgoing edge, then the output label of this edge is the empty word. The transducer $\mathcal{B}(AD)$ works as follows on an input string w. Consider the (unique) path in $\mathcal{B}(AD)$ corresponding to w. The letters of w that correspond to an edge that is the unique outgoing edge of a given state are erased; other letters are unchanged.

We can then state the following theorem.

Theorem 2: Algorithm ENCODER can be realized by a sequential transducer (generalized sequential machine).

Concerning the algorithm DECODER, remark (see Section II) that the function γ is not injective and that we need some additional information, for instance the length of the original uncompressed word, in order to reconstruct it without ambiguity. Therefore, DECODER can be realized by the same transducer as above, by interchanging input and output labels (denote it by $B'(AD)$), with a supplementary instruction to stop the decoding.

Let $Q = Q_1 \cup Q_2$ be a partition of the set of states Q, where Q_j is the set of states having j outgoing edges ($j = 1, 2$). For any $q \in Q_1$, define $p(q) = (q, q_1, \ldots, q_r)$ as the unique path in the transducer for which $q_h \in Q_1$ for $h < r$ and $q_r \in Q_2$.

Given an input word $v = b_1 b_2 \ldots b_m$, there exists in $B'(AD)$ a unique path $i, q_1, \ldots, q_{m'}$ such that $q_{m'-1} \in Q_2$ and the transition from $q_{m'-1}$ to $q_{m'}$ correspond to the input letter b_m. If $q_{m'} \in Q_2$, then the output word corresponding to this path in $B'(AD)$ is the unique word w such that $\gamma(w) = v$. If $q_{m'} \in Q_1$, then we can stop the decoding algorithm realized by $B'(AD)$ in any state $q \in p(q_{m'})$, and, for different states, we obtain different decodings. So we need supplementary information (for instance, the length of the original uncompressed word) to perform the decoding. In this sense we can say that $B'(AD)$ realizes sequentially the algorithm DECODER (cf. also [17]).

The constructions and the results given above on finite antidictionaries and transducers can be generalized also to the case of rational antidictionaries, or, equivalently, when the set of words "produced by the source" is a regular (rational) language. In these cases it is not, in a strict sense, necessary to introduce explicitly antidictionaries and all the methods can be presented in terms of automata and transducers, as above. Remark however that the presentation given in Section II in terms of antidictionaries is more general, since it includes the non rational case. Moreover, even in the finite case, the construction of automata and transducers from a fixed text, given in the next section, makes an explicit use of the notion of minimal forbidden words and of antidictionaries.

B. A Synchronization Property

In the sequel we prove a synchronization property of automata built from finite antidictionaries, as described above. This property also "characterizes" in some sense finite antidictionaries. This property is a classical one and it is of fundamental importance in practical applications.

Definition 1: Given a deterministic finite automaton $\mathcal{A}$, we say that a word $w = a_1 \cdots a_k$ is synchronizing for $\mathcal{A}$ if, whenever w represents the label of two paths $(q_1, a_1, q_2) \cdots (q_k, a_k, q_{k+1})$ and $(q'_1, a_1, q'_2) \cdots (q'_k, a_k, q'_{k+1})$ of length k, then the two ending states q_{k+1} and q'_{k+1} are equal.

If $L(\mathcal{A})$ is factorial, any word that does not belong to $L(\mathcal{A})$ is synchronizing. Clearly in this case synchronizing words in $L(\mathcal{A})$ are much more interesting. Remark also that, since $\mathcal{A}$ is deterministic, if w is synchronizing for $\mathcal{A}$, then any word $w' = wv$ that has w as prefix is also synchronizing for $\mathcal{A}$.

Definition 2: A deterministic finite automaton $\mathcal{A}$ is local if there exists an integer k such that any word of length k is synchronizing. Automaton $\mathcal{A}$ is also called k-local.
Remark that if $\mathcal{A}$ is k-local then it is m-local for any $m \geq k$.

Given a finite antifactorial language AD, let $\mathcal{A}(AD)$ be the automaton associated with AD that recognizes the language $L(AD)$. Let us eliminate the sink states and edges going to them. Since there is no possibility of misunderstanding, we denote the resulting automaton by $\mathcal{A}(AD)$ again. Notice that it has no sink state, that all states are terminal, and that $L(\mathcal{A}(AD))$ is factorial.

Theorem 3: Let AD be a finite antifactorial antidictionary and let k be the length of the longest word in AD. Then automaton $\mathcal{A}(AD)$ associated to AD is $(k-1)$-local.

Proof: Let $u = a_1 \cdots a_{n-1}$ be a word of length $n-1$. We have to prove that u is synchronizing. Suppose that there exist two paths $(q_1, a_1, q_2) \cdots (q_{n-1}, a_{n-1}, q_n)$ and $(q_1', a_1, q_2') \cdots (q_{n-1}', a_{n-1}, q_n')$ of length $n-1$ labeled by u. We have to prove that the two ending states q_n and q_n' are equal. Recall that states of $\mathcal{A}$ are words, and, more precisely they are the proper prefixes of words in AD. A simple induction on i, $1 \leq i \leq n$ shows that q_i (respectively q_i') "is" the longest suffix of the word $q_1 a_1 \cdots a_i$ (respectively $q_1' a_1 \cdots a_i$) that is also a "state", *i.e.*, a proper prefix of a word in AD. Hence q_n (respectively q_n') is the longest suffix of the word $q_1 u$ (respectively $q_1' u$) that is also a proper prefix of a word in AD. Since all proper prefixes of words in AD have length at most $n-1$, both q_n and q_n' have length at most $n-1$. Since u has length $n-1$, both they are the longest suffix of u that is also a proper prefix of a word in AD, *i.e.*, they are equal. ■

In other terms, the theorem says that only the last $k-1$ bits matter for determining whether AD is avoided or not. The theorem admits a "converse" that shows that locality characterizes in some sense finite antidictionaries (cf. Propositions 2.8 and 2.14 of [18]).

Theorem 4: If automaton $\mathcal{A}$ is local and $L(\mathcal{A})$ is a factorial language then there exists a finite antifactorial language AD such that $L(\mathcal{A}) = L(AD)$.

Let AD be an antifactorial antidictionary and let k be the length of the longest word in AD. Let also $w = w_1 u v w_2 \in L(AD)$ with $|u| = k-1$ and let $\gamma(w) = y_1 y_2 y_3$ be the word produced by our encoder of Section II with input AD and w. The word y_1 is the word produced by our encoder after processing $w_1 u$, the word y_2 is the word produced by our encoder after processing v and the word y_3 is the word produced by our encoder after processing w_2.

The proof of next theorem is an easy consequence of previous definitions and of the statement of Theorem 3.

Theorem 5: The word y_2 depends only on the word uv and it does not depend on the contexts of it, w_1 and w_2.

The property stated in the theorem has an interesting consequence for the design of pattern matching algorithms on words compressed by the algorithm ENCODER. It implies that to search the compressed word for a pattern, it is not necessary to decode the whole word. Just a limited left context of an occurrence of the pattern needs to be processed. The same property allows the design of highly parallelizable compression algorithms. The idea is that the compression can be performed independently and in parallel on any block of data. If the text to be compressed is parsed into blocks of data in such a way that each block overlaps the next block by a length not smaller than the length of the longest word in the antidictionary, then it is possible to run the whole compression process in parallel.

IV. EFFICIENCY

In this section we evaluate the efficiency of our compression algorithm relatively to a source corresponding to the finite antidictionary AD.

Indeed, the antidictionary AD naturally defines a source $S(AD)$ in the following way. Let $\mathcal{A}(AD)$ be the automaton constructed in the previous section with no sink states and recognizing the factorial language $L(AD)$ (all states are terminal). To avoid trivial cases, we suppose that in this automaton all the states have at least one outgoing edge. Recall that since our algorithms work on a binary alphabet, all states have at most two outgoing edges.

For any state of $\mathcal{A}(AD)$ with only one outgoing edge we give to this edge probability 1. For any state of $\mathcal{A}(AD)$ with two outgoing edge we give to these edges probability 1/2. This defines a deterministic (or unifilar, cf. [19]) Markov source, denoted $S(AD)$. Notice also that, by Theorem 3, that $S(AD)$ is a Markov source of finite order or finite memory (cf. [19]). We call a binary Markov source with this probability distribution an *balanced source*.

Remark that our compression algorithm is defined exactly for all the words "emitted" by $S(AD)$.

In what follows we suppose that the graph of the source S, *i.e.*, the graph of automaton $\mathcal{A}(AD)$, is strongly connected. The results that we prove can be extended to the general case by using standard techniques of Markov Chains (cf. [19], [20], [21] and [22]). Recall (cf. Theorem 6.4.2 of [19]) that the entropy $H(S)$ of a deterministic Markov source S is $H(S) = -\Sigma_{i,j=1}^{n} \mu_i \gamma_{i,j}\ log_2(\gamma_{i,j})$, where $(\gamma_{i,j})$ is the stochastic matrix of S and $(\mu_1, \cdots, \mu_n)$ is the stationary distribution of S.

We now state three lemmas.

Lemma 1: The entropy of a balanced source S is given by $H(S) = \Sigma_{i \in D} \mu_i$ where D is the set of all states that have two outgoing edges.

Proof: By definition

$$H(S) = -\Sigma_{i,j=1}^{n} \mu_i \gamma_{i,j}\ log_2(\gamma_{i,j}).$$

If i is a state with only one outgoing edge, by definition this edge must have probability 1. Then $\Sigma_j \mu_i \gamma_{i,j}\ log_2(\gamma_{i,j})$ reduces to $\mu_i\ log_2(1)$, that is equal to 0. Hence

$$H(S) = -\Sigma_{i \in D} \Sigma_{j=1}^{n} \mu_i \gamma_{i,j}\ log_2(\gamma_{i,j}).$$

Since from each $i \in D$ there are exactly two outgoing edges having each probability $1/2$, one has

$$H(S) = -\Sigma_{i \in D} 2\mu_i (1/2)\ log_2(1/2) = \Sigma_{i \in D} \mu_i$$

as stated. ■

Lemma 2: Let $w = a_1 \cdots a_m$ be a word in $L(AD)$ and let $q_1 \cdots q_{m+1}$ be the sequence of states in the path determined by w in $\mathcal{A}(AD)$ starting from the initial state. The length of $\gamma(w)$ is equal to the number of states q_i, $i = 1, \ldots, m$, that belong to D, where D is the set of all states that have two outgoing edges.

Proof: The statement is straightforward from the description of the compression algorithm and the implementation of the antidictionary with automaton $\mathcal{A}(AD)$. ■

Through a well-known results on "large deviations" (cf. Problem IX.6.7 of [23]), we get a kind of optimality of the compression scheme.

Let $\mathbf{q}= q_1, \cdots q_m$ be the sequence of m states of a path of $\mathcal{A}(AD)$ and let $L_{m,i}(\mathbf{q})$ be the frequency of state q_i in this sequence, *i.e.*, $L_{m,i}(\mathbf{q}) = m_i/m$, where m_i is the number of occurrences of q_i in the sequences $\mathbf{q}$. Let also $X_m(\epsilon) = \{ \mathbf{q} \mid \mathbf{q} \text{ has } m \text{ states and } max_i\ |L_{m,i}(\mathbf{q}) - \mu_i| \geq \epsilon\}$, where $\mathbf{q}$ represents a sequence of m states of a path in $\mathcal{A}(AD)$. In other words, $X_m(\epsilon)$ is the set of all sequences of states representing path in $\mathcal{A}(AD)$ that "deviate" at least of ϵ in at least one state q_i from the theoretical frequency μ_i.

Lemma 3: For any $\epsilon > 0$, the set $X_m(\epsilon)$ satisfies the equality $lim \frac{1}{m} log_2 Pr(X_m(\epsilon)) = -c(\epsilon)$, where $c(\epsilon)$ is a positive constant depending on ϵ.

We now state the main theorem of this section. The proof of it uses the three previous lemmas. It states that for any ϵ the probability that the compression rate $\tau(v) = |\gamma(v)|/|v|$ of a string of length n is greater than $H(S(AD))+\epsilon$, goes exponentially to zero. Hence, as a corollary, almost surely the compression rate of an infinite sequence emitted by $S(AD)$ reaches the entropy $H(S(AD))$, that is the best possible result.

Theorem 6: Let $K_m(\epsilon)$ be the set of words w of length m such that the compression rate $\tau(v) = |\gamma(v)|/|v|$ is greater than $H(S(AD))+\epsilon$. For any $\epsilon > 0$ there exist a real number $r(\epsilon)$, $0 < r(\epsilon) < 1$, and an integer $\overline{m}(\epsilon)$ such that for any $m > \overline{m}(\epsilon)$, $Pr(K_m(\epsilon)) \leq r(\epsilon)^m$.

Proof: Let w be a word of length m in the language $L(AD)$ and let $q_1, \cdots, q_{m+1}$ be the sequence of states in the path determined by w in $\mathcal{A}(AD)$ starting from the initial state. Let $\mathbf{q}= (q_1, \cdots, q_m)$ be the sequence of the first m states. We know, by Lemma 2, that the length of $\gamma(w)$ is equal to the number of states q_i, $i = 1 \cdots m$, in $\mathbf{q}$ that belong to D, where D is the set of all states having two outgoing edges.

If w belongs to $K_m(\epsilon)$, *i.e.*, if the compression rate $\tau(v) = |\gamma(v)|/|v|$ is greater than $H(S(AD)) + \epsilon$, then there must exists an index j such that $L_{m,j}(\mathbf{q}) > \mu_j + \epsilon/|D|$. In fact, if for all j, $L_{m,j}(\mathbf{q}) \leq \mu_j + \epsilon/|D|$ then, by definitions and by Lemma 1,

$$\tau(v) = \Sigma_{j \in D} L_{m,j}(\mathbf{q}) \leq \Sigma_{j \in D} \mu_j + \epsilon = H(S(AD)) + \epsilon,$$

a contradiction. Therefore the sequence of states $\mathbf{q}$ belongs to $X_m(\epsilon/d)$. Hence $Pr(K_m(\epsilon)) \leq Pr(X_m(\epsilon/d))$.

By Lemma 3, there exists an integer $\overline{m}(\epsilon)$ such that for any $m > \overline{m}(\epsilon)$ one has

$$\frac{1}{m} log_2 Pr(X_m(\frac{\epsilon}{d})) \leq -\frac{1}{2} c(\frac{\epsilon}{d}).$$

Then $Pr(K_m(\epsilon)) \leq 2^{-(1/2)c(\epsilon/d)m}$. If we set $r(\epsilon) = 2^{-(1/2)c(\epsilon/d)}$, the statement of the theorem follows. ■

Theorem 7: The compression rate $\tau(\mathbf{x})$ of an infinite sequence $\mathbf{x}$ emitted by the source $S(AD)$ reaches the entropy $H(S(AD))$ almost surely.

V. How to build Antidictionaries

In practical applications the antidictionary might not be given *a priori* but it must be derived either from the text to be compressed or from a family of texts belonging to the assumed source of the text to be compressed.

There exist several criteria to build efficient antidictionaries, depending on different aspects or parameters that one wishes to optimize in the compression process. Each criterion gives rise to different algorithms and implementations.

All our methods to build antidictionaries are based on data structures to store factors of words, such as suffix tries, suffix trees, DAWGs, and suffix and factor automata (see for instance Theorem 15 in [10]). In these structures, it is possible to consider a notion of suffix link. This link is essential to design efficient algorithms to build representations of sets of minimal forbidden words in term of tries or trees. This approach leads to construction algorithms that run in linear time in the length of the text to be compressed.

A rough solution to control the size of antidictionaries is obviously to bound the length of words in the antidictionary. A better solution in the static compression scheme is to prune the trie of the antidictionary with a criterion based on the tradeoff between the space of the trie to be sent and the gain in compression, this will be developed in next section. However, the first solution is enough to get compression rates that reach asymptotically the entropy for balanced sources, even if this is not true for general sources. Both solutions can be designed to run in linear time.

We present in this section a very simple construction to build finite antidictionaries of a finite word w. It is the base on which several variations are developed. The idea is to build the automaton accepting the words having same factors of w of length k and, from this, to build the set of minimal forbidden words of length k of the word w. It can be used as a first step to build antidictionaries for fixed sources. In this case our scheme can be considered as a step for a compressor generator (compressor compiler). In the design of a compressor generator, or compressor compiler, statistical considerations and the possibility of making "errors" in predicting the next letter play an important role, as discussed in Section VII.

Algorithm BUILD-AD described hereafter builds the set of minimal forbidden words of length k ($k > 0$) of the word w. It takes as input an automaton accepting the words that have the same factors of length k (or less) as w, *i.e.*, accepting the language

$$L_k = \{x \in \{0,1\}^* \mid (u \in F(x) \text{ and } |u| \leq k) \Rightarrow u \in F(w)\}.$$

The preprocessing of the automaton is done by the algorithm BUILD-FACT whose central operation is described by the function NEXT.

```
BUILD-FACT (word w ∈ {0,1}*, integer k > 0)
1.  i ← new state; Q ← {i};
2.  level(i) ← 0;
3.  p ← i;
4.  while not end of string w
5.      a ← next letter of w;
6.      p ← NEXT(p, a, k);
7.  return trie (Q, i, Q, δ), function f;
```

```
NEXT (state p, letter a, integer k > 0)
1.  if δ(p, a) defined
2.      return δ(p, a);
3.  else if level(p) = k
4.      return NEXT(f(p), a, k);
5.  else
6.      q ← new state; Q ← Q ∪ {q};
7.      level(q) ← level(p) + 1;
8.      δ(p, a) ← q;
9.      if (p = i) f(q) ← i;
10.     else f(q) ← NEXT(f(p), a, k);
11.     return q;
```

```
BUILD-AD (trie (Q, i, Q, δ), function f, integer k > 0)
1.  T ← ∅; δ' ← δ;
2.  for each p ∈ Q, 0 < level(p) < k, in breadth-first order
3.      for a ← 0 then 1
4.          if δ(p, a) is undefined and δ(f(p), a) is defined
5.              q ← new state; T ← T ∪ {q};
6.              δ'(p, a) ← q;
7.  Q ← Q \ {states of Q from which no δ'-path leads to T}
8.  return trie (Q ∪ T, i, T, δ');
```

The automaton is represented by both a trie and its failure function f. If p is a node of the trie associated with the word av, $v \in \{0,1\}^*$ and $a \in \{0,1\}$, $f(p)$ is the node associated with v. This is a standard technique used in the construction of suffix trees (see [24] for example). It is used here in algorithm BUILD-AD (line 4) to test the minimality of forbidden words according to the equality $MF(L) = AL \cap LA \cap (A^* \setminus L)$.

The above construction gives rise to the following *static* compression scheme in which we need to read the text twice, the first time to construct the antidictionary AD and the second time to encode the text.

Informally, the encoder sends a message z of the form $(x, y, \sigma(n))$ to the decoder, where x is a description of the antidictionary AD, y is the text coded according to AD, as described in Section II, and $\sigma(n)$ is the usual binary code of the length n of the text. The decoder first reconstructs from x the antidictionary and then decodes y according to the algorithm in Section II. The antidictionary AD is composed in this simple compression scheme by all minimal forbidden words of length k of w, but other intelligent choices of subsets of AD are possible. We can describe the antidictionary AD for instance by coding with standard techniques the trie associated with AD to obtain the word x. A basic question is how fast must grow the number k as function of the length n of the word w. In this simple compression scheme we choose k to be any function such that one has that $|x| = o(n)$, but other choices are possible. Since the compression rate is the size $|z|$ of z divided by the length n of the text, we have that $|z|/n = |y|/n + o(n)$. Assuming that for n and k large enough the source $S(AD)$, as in Section IV, approximates the source of the text, then, by the results of Section IV, the compression rate is "optimal".

For instance, suppose that w is emitted by an balanced Markov source S with memory h, and let L be the formal language composed of all finite words that can be emitted by S. By Theorem 4 there exists a finite antifactorial language N such that $L = L(N)$. Moreover, since S has memory h, the words in N have length smaller than or equal to $h+1$. If $|w|$ is such that $k > h$ then AD contains N and, therefore $H(S(AD)) \leq H(S(N)) = H(S)$. By Corollary 1 we can deduce that this simple compression scheme turns out to be universal for the family of balanced Markov sources with finite memory (cf. [25]).

Let $\mathbf{w} = a_1 a_2 \cdots$ be a binary infinite word that is periodic (*i.e.*, there exists integer $P > 0$ such that for any index i the letter a_i is equal to the letter a_{i+P}), and let w_n be the prefix of $\mathbf{w}$ of length n. We want to compress the word w_n following our simple scheme informally described above.

It is not difficult to prove that the compression rate for w_n is $|z|/n = O(\sigma(n)) = O(log_2(n))$, which means that the scheme can achieve an exponential compression.

VI. PRUNING ANTIDICTIONARIES

In this section, as well as in previous section, we consider a *static* compression scheme in which we need to read the text twice: the first time to construct the antidictionary AD and the second time to encode the text.

In this section, however, we suppose that we have enough resources to build, in linear time, a suffix or a factor automaton (or their compacted version, cf. [26]) of the finite text string to be compressed. From these structures we can obtain in linear time a trie representating of *all* minimal forbidden words of the text (cf. [10]). It can be shown that the total length of all minimal forbidden words can be quadratic in the size of the original text. However the trie representing these words is of linear size. It is clear that if we want to get good compression ratios not all the minimal forbidden words should be considered.

The first idea developed in this section is to prune the trie of the antidictionary with some criteria based on the tradeoff between the space of the trie to be sent and the gain in compression. Clearly, the space of the trie to be sent strictly depend on how we encode the trie.

Using a classical approach, in this section we recall that a binary tree that has k nodes can be encoded using two bits for each node, which gives $2k$ bits for the whole tree. Indeed, depending on whether a subtree S of a binary tree T has both subtrees, only the right subtree, only the left subtree, or no subtree, the root of S can be encoded respectively by the strings 11, 10, 01, 00. This is done recursively in a prefix traversal of the whole tree. All the results p-resented in this section can be easily extended to the case when a node of the trie can be encoded using α bits for each node, where α is a positive real number.

The second idea presented afterwards is to compress the words retained in the antidictionary using the antidictionary itself.

The two operations, pruning and self compressing, can be applied iteratively on antidictionaries. They lead to very compact representations of antidictionaries, producing higher compression ratios.

A. Pruned Antidictionary

A linear-time algorithm for obtaining the trie $\mathcal{T}$ of all minimal forbidden word of a fixed text t can be found in [10]. Hence we suppose here that we have this trie $\mathcal{T}$.

In order to make a tradeoff between the space of the trie to be sent and the gain in compression, we have to know how much each forbidden word contributes to the compression. Minimal forbidden words of text t correspond in a bijective way to the leaves of the trie $\mathcal{T}$, *i.e.* with any leave q of the tree we can associate the corresponding minimal forbidden word $w(q)$. Indeed if we identify, as in Section III, the nodes of the trie $\mathcal{T}$ to the prefixes of the minimal forbidden words, then the function w is the identity.

We define a *cost function* c that associates with any leaf q of $\mathcal{T}$ the number of bits $c(q)$ that the word $w(q)$ contributes to erase during the compression of the text t. This number $c(q)$ is also the number of times that the longest proper prefix of $w(q)$ appears in text t as a factor but not as a suffix. In another words, the number $c(q)$ is the number of times that a state p is traversed while reading the text t in the automaton $\mathcal{A}(AD)$, where p leads to state q by some letter a (cf. Section III and Theorem 1). Indeed the last letter of the text is not considered in this process because there is nothing to erase after it. By Theorem 1, the function c can be computed in linear time.

We further define the *gain* (saving) of a subtree S of the trie $\mathcal{T}$ representing an antidictionary T as $g(S) = \Sigma(c(q) \mid q \text{ leaf of } S) - 2m_S$ where m_S is the number of nodes of S.

Indeed the number of bits that have to be sent after compression is composed of: $2\lfloor \log n \rfloor$ bits to encode the length n of the text t (cf. the cascading lengths technique in [4] and references therein); $2m_{\mathcal{T}}$ bits for a description of the antidictionary $\mathcal{T}$; $|\gamma(t)|$ bits for the text compressed using $\mathcal{T}$. The overall size is

$$2\lfloor \log n \rfloor + 2m_{\mathcal{T}} + |\gamma(t)| = 2\lfloor \log n \rfloor + n - g(\mathcal{T})$$

by definition of $g(\mathcal{T})$.

Since $2\lfloor \log n \rfloor + n$ is fixed and since the gain $g(\mathcal{T})$ is the sum of the gain of its subtrees minus 2 bits (for encoding the root), then pruning subtrees of $\mathcal{T}$ that have a negative gain increases the gain of $\mathcal{T}$ and, consequently, decreases the overall number of bits that have to be sent after compression.

Suppose however that S_2 is a subtree of S_1 which is, in turn, a subtree of the trie $\mathcal{T}$. Suppose further that S_2 has a negative gain and the same holds for S_1, but that S_1 has a positive gain if S_2 is pruned from it. In this case, in order to obtain better compression ratios, the best thing to do is to prune S_2 and not the whole S_1. It is thus natural to consider the optimization problem related to an abstact non-negative function c (defined on leaves of $\mathcal{T}$) where one instance is a trie $\mathcal{T}$ representing a prefix code C, and a solution is a trie $\mathcal{T}'$ that represents a subset of C and that maximizes the gain $g(\mathcal{T}')$.

In what follows we show that a bottom-up approach gives a linear-time solution to this problem.

With any subtree S of $\mathcal{T}$ we associate the function g', called the *pruned gain*, that is defined by

$$g'(S) = \begin{cases} 0 & \text{if } S \text{ is empty} \\ c(S) - 2 & \text{if } S \text{ is a leaf} \\ g'(S_1) - 2 & \text{if } S \text{ has one child } S_1 \\ M & \end{cases}$$

where $M = \max(g'(S_1), g'(S_2), g'(S_1) + g'(S_2)) - 2$, with S_1 and S_2 children of S.

From the above definition it is not difficult to see that it is possible to compute function g' in linear time with respect to the size of the trie $\mathcal{T}$, in a bottom-up traversal of the trie.

We can now present the simple pruning algorithm.

```
SIMPLE PRUNING (trie T, function c)
1.  compute g'(S) for each subtree S of T;
2.  eliminate subtrees S of T for which g'(S) ≤ 0;
3.  return modified trie T;
```

The following proposition is a consequence of the descriptions given above, and the next theorem shows that the output of the algorithm gives a solution to the optimization problem described above.

Proposition 1: Algorithm SIMPLE PRUNING can be performed in linear time.

Theorem 8: Let $\mathcal{T}$ be a trie representing a prefix code C and let c be a non-negative function defined on leaves of $\mathcal{T}$. The output $\mathcal{T}'$ of algorithm SIMPLE PRUNING represents a subset of C and $g'(\mathcal{T}')$ is maximum. Moreover we have that $g(\mathcal{T}') = g'(\mathcal{T}')$.

Proof: First of all we claim that the trie $\mathcal{T}'$ output by algorithm SIMPLE PRUNING represents a subset of C. Indeed, by the definition of g' it follows that if a subtree S of $\mathcal{T}$ is not a leaf and if $g'(S) > 0$, then S must have at least one child S_1 with positive pruned gain, *i.e.* $g'(S_1) > 0$. This fact implies that all leaves of $\mathcal{T}'$ are leaves of $\mathcal{T}$, proving the claim.

The rest of the proof is done by induction on the height of $\mathcal{T}$. If $\mathcal{T}$ is empty there is nothing to prove. If $\mathcal{T}$ has height 0 then $\mathcal{T}$ is a leaf and we already have $g(\mathcal{T}) = g'(\mathcal{T})$. If $g(\mathcal{T}) > 0$, $\mathcal{T}$ itself is equal to $\mathcal{T}'$, otherwise $\mathcal{T}'$ is the empty tree. In both cases the statement of the theorem is satisfied.

Suppose now that $\mathcal{T}$ has height > 0. Either it has just one child S_1 or it has two children S_1 and S_2.

Suppose that $\mathcal{T}$ has two children S_1 and S_2. S_i; they are both tries and we can associate to them the restriction of the function gain to all subtrees. By applying algorithm SIMPLE PRUNING with input S_i, $i = 1, 2$, and function c (restricted to leaves of corresponding subtrees), we obtain as output a modified trie S_i'. By induction we know that $g(S_i') = g'(S_i')$ and that this value maximizes the function gain. Therefore, if both $g(S_1')$ and $g(S_2')$ are positive, a trie $\mathcal{T}'$ representing a subset of C and maximizing the function gain is the trie that has the same root as $\mathcal{T}$ and has children S_1' and S_2'. Moreover $g(\mathcal{T}') = g'(\mathcal{T}')$ and algorithm SIMPLE PRUNING does not prune S_1 and S_2 from $\mathcal{T}'$ so the theorem is proved in this case.

The other cases, ($g(S_1) \leq 0$ and $g(S_2) > 0$), ($g(S_1) > 0$ and $g(S_2) \leq 0$), ($g(S_1) \leq 0$ and $g(S_2) \leq 0$), and the case when $\mathcal{T}$ has only one child S_1 are dealt in analogous manner. ∎

Remark that the statement of Theorem 8 holds essentially because pruning a subtree S of $\mathcal{T}$ does not affect the value of function gain over all other subtrees of $\mathcal{T}$. This fact is not true anymore with the self-compressing approach used in next subsection.

B. Self-compressing the antidictionary

Let *AD* be an antifactorial antidictionary for text t. Since *AD* is antifactorial then, for any $v \in AD$ the set $AD \setminus \{v\}$ is an antidictionary for v. Therefore it is possible to compress v using $AD \setminus \{v\}$ or a subset of it.

One can think of a strategy that sends to the decoder, in a static approach, all words v of *AD* compressed by algorithm ENCODER with a subset of $AD \setminus \{v\}$ and v as input. This would achieve better compression. We call this approach *self-compression*; it is the subject of this subsection.

Let us first try to compress any word $v \in AD$ by using the whole $AD \setminus \{v\}$ and let us denote by $\gamma_1(v)$ the compressed version of v by using $AD \setminus \{v\}$. Notice that the words of *AD* that are used in compressing v have length $\leq |v|$. Further, if $u \in AD$ with $|u| = |v|$ is used to erase the last letter of v, then u must coincide with v except for the last letter, that is, $u = xa$, $v = xb$ and $a \neq b$. In addition it is easy to see that $\gamma_1(u) = \gamma_1(v)$. This word is also equal to $\gamma_1(x)$ that has been compressed by using the antidictionary of all words of *AD* having length shorter than $|v| = |u|$.

As as a special case of the next proposition, a set $\{u, v\}$ having these properties can occur at most once in any antidictionary *AD* of a text t.

A pair of words (v, v_1) is called *stopping pair* if $v = ua, v_1 = u_1 b \in AD$, with $a, b \in \{\mathtt{0}, \mathtt{1}\}$, $a \neq b$, and u is a suffix of u_1.

Proposition 2: Let *AD* be an antifactorial antidictionary of a text t. If there exists a stopping pair (v, v_1) with $v_1 = u_1 b$, $b \in \{\mathtt{0}, \mathtt{1}\}$, then u_1 is a suffix of t and does not appear elsewhere in t. Moreover there exists at most one pair of words having these properties.

Proof: Since $u_1 b \in AD$, u_1 is a factor of t. Suppose that $u_1 c$ appears as a factor of t, with $c \in \{\mathtt{0}, \mathtt{1}\}$. Since u is a suffix of u_1, letter c is not letter a (because ua is forbidden) and is not letter b (because $u_1 b$ is forbidden), a contradiction. Hence u_1 is a suffix of t and does not appear elsewhere in t.

Since u_1 is a suffix of t, then also u is a suffix of t. Suppose that there exists another pair $(v' = u'c, v_1' = u_1' d) \neq (v, v_1)$ of words in *AD* with $c, d \in \{\mathtt{0}, \mathtt{1}\}$, $a \neq b$, and u' is a suffix of u_1'. Then u_1' and u' are also suffixes of t and it is not difficult to prove by cases that one of the four words among v, v_1, v', v_1' is a factor of another, contradicting the antifactoriality of *AD*. ∎

Let us suppose now that $v_1, \ldots, v_k$ is a sequence of all words in *AD* such that for any i, $1 \leq i \leq k-1$, $|v_i| \leq |v_{i+1}|$. If one knows that there exists no v_j such that $|v_j| = |v_i|$ and v_j has been used to erase the last letter of v_i, then the set $AD_1 = \{v_1, \ldots v_{i-1}\}$ is the antidictionary used for compressing v_i to get $\gamma_1(v)$, and v_i can be recovered from both $\gamma(v_i)$ and $|v_i|$ using algorithm DECODER. If there exists v_j such that $|v_j| = |v_i|$ and v_j has been used to erase the last letter of v_i then the set $AD_1 = \{v_1, \ldots v_{i-1}\}$ is the antidictionary used for obtaining the compressed version $\gamma_1(x) = \gamma_1(v_i)$ of the longest common prefix x of v_i and v_j, with $|x| = |v_i| - 1$. Also in this case x and therefore v_i and v_j, can be recovered from both $\gamma_1(x) = \gamma_1(v_i)$ and $|x| = |v_i| - 1$ using algorithm DECODER.

By the above discussion, it follows that if one knows the sequence $(\gamma_1(v_1), |v_1|), (\gamma_1(v_2), |v_2|), \ldots, (\gamma_1(v_k), |v_k|)$, together with the couple (i, j) such that v_i and v_j have been used to mutually erase their last letter ($i = j = 0$ if there is no such a pair), then the decoder can reconstruct, in this order, words $v_1, v_2, \ldots, v_k$. That is, decoder can reconstruct the whole antidictionary *AD*.

Unfortunately, while *AD*, being antifactorial, is also a prefix code and can be represented by a trie, this is not true anymore for the set $X_1 = \{\gamma_1(v) \mid v \in AD\}$. For example, the reader can easily verify that if $AD = \{\mathtt{11}, \mathtt{000}, \mathtt{10101}, \mathtt{00100100}, \mathtt{1010010100101}\}$ then $X_1 = \{\mathtt{11}, \mathtt{000}, \mathtt{111}, \mathtt{0000}, \mathtt{1111}, \}$. Also, if $AD = \{\mathtt{10}, \mathtt{110}, \cdots, \mathtt{1}^{\mathtt{n}}\mathtt{0}\}$ then, for any $n \geq 0$, $X_1 = \{10\}$. Consequently the space saved by self compressing the antidictionary could be lost in encoding the set X_1.

We propose a different approach that makes use of the same idea and leads to simple algorithms for self-compressing and recovering the antidictionary *AD*. These algorithms run in linear time in the size of the trie $\mathcal{T}$ representing the antifactorial antidictionary AD and, moreover, the compression ratios obtained with the pruning technique can only be improved by the next self compression technique.

We present a formal description of the technique. Given a word $v \in AD$, we compress it using an antidictionary AD' that dynamically changes at any step of the **while** loop on line 2 of algorithm ENCODER. While dealing with a proper prefix u of v and the letter a following it, the antidictionary AD' is composed of all words belonging to AD with length not greater than $|u|$. Letter a is erased if and only if there exists a word $u'b \in AD$, $b \neq a$, with u' a proper suffix of u. Let us call $\gamma_2(v)$ the compressed version of v obtained in this way and let $X_2 = \{\gamma_2(v) \mid v \in AD\}$.

This kind of self-compression can be performed in linear time by next algorithm SELF-COMPRESS. It has as input both the trie $\mathcal{T}$ that represents AD and the function δ of automaton $\mathcal{A}(AD)$ (cf. algorithm L-AUTOMATON). Notice that δ is defined on nodes of $\mathcal{T}$. Its output $\mathcal{T}'$ is the trie accepting the set $X_2 = \{\gamma_2(v) \mid v \in AD\}$. The algorithm performs breadth-first traversal of $\mathcal{T}$ implemented by the queue $\mathcal{Q}$. During the traversal, it creates a self-compressed version $\mathcal{T}'$ of $\mathcal{T}$ that represents the set X_2.

SELF-COMPRESS (trie $\mathcal{T}$, function δ))
1. $i \leftarrow$ root of $\mathcal{T}$;
2. create root i';
3. add (i, i') to empty queue $\mathcal{Q}$;
4. **while** $\mathcal{Q} \neq \emptyset$
5. extract (p, p') from $\mathcal{Q}$;
6. **if** q_0 and q_1 are children of p
7. create q_0' and q_1' as children of p';
8. add (q_0, q_0') and (q_1, q_1') to $\mathcal{Q}$;
9. **else if** q is a unique child of p and $q = \delta(p, a)$, $a \in A$
10. **if** $\delta(p, \neg a)$ is a leaf
11. add (q, p') to $\mathcal{Q}$;
12. **else** create q' as a-child of p';
13. add (q, q') to $\mathcal{Q}$;
14. **return** trie having root i';

The correctness of algorithm SELF-COMPRESS relies on the following proposition and the discussion thereafter.

Proposition 3: If a node p in the trie $\mathcal{T}$ has two children q_0 and q_1 then its corresponding node p' in the output trie $\mathcal{T}'$ also has two children.

Proof: If q_0 and q_1 are both leaves, they represent two minimal forbidden words ua and ub, $a \neq b$. There is no minimal forbidden words in the form $u'a$ or $u'b$ with u' a proper suffix of u because AD is antifactorial. Therefore neither letter a nor letter b can be erased by the technique.

If q_0 and q_1 are not leaves, they represent two words ua and ub, $a \neq b$, that are factors of text t. There is no minimal forbidden words in the form $u'a$ or $u'b$ with u' a proper suffix of u because these words are also factors of t. Therefore neither letter a nor letter b can be erased by the technique.

Let us suppose now that only one node among q_0 and q_1 is a leaf. For instance, let us assume that q_0 is a leaf and q_1 is not a leaf. They represent respectively two words ua and ub, $a \neq b$. Letter a cannot be erased because in the antidictionary there is no word in the form $u'b$ with u' a proper suffix of u, ub being a factor of t. Letter b cannot be erased because in the antidictionary there is no word in the form $u'a$ with u' a proper suffix of u, since ua is in the antidictionary and the antidictionary is antifactorial. ■

The previous proposition explains why the algorithm creates two nodes q_0' and q_1' at line 7.

We next consider lines 10–13, in which node p of $\mathcal{T}$ has only one child $q = \delta(p, a)$. The node $\delta(p, \neg a)$ cannot have higher level than p because p has only one child. Hence, letter a is erased if and only if $\delta(p, \neg a)$ is a leaf, by definition of the technique.

Finally, if p has no children, *i.e.* p is a leaf, nothing is done by the algorithm but extracting (p, p') from the queue.

Corollary 1: Tries $\mathcal{T}$ and $\mathcal{T}'$ have the same number of internal nodes that have two children and, consequently, have the same number of leaves. Trie $\mathcal{T}'$ represents the prefix code X_2.

The corollary implies that $X_2 = \{\gamma_2(v) \mid v \in AD\}$ can be uniquely reconstructed from $\mathcal{T}'$. There is an additional property that allows reconstructing AD from X_2 without considering lengths of words in AD. This simplifies the procedure. The next proposition follows readily from definitions.

Proposition 4: If there exists no stopping pair in AD then for any $v \in AD$, the last letter of v is not erased during the self-compression to get $\gamma_2(v)$.

If the decoder has the additional information that the last letter of t was not erased at compression time then it can use this fact as a stop criterion. This is also possible even if the antidictionary changes dynamically. Indeed the decoder just has to stop after processing the last letter of the compressed text. Therefore there is no need to use the length of the text to stop decoding.

To ensure that the last letter of any $v \in AD$ is not erased and to meet the above hypothesis, it is sufficient to eliminate the only possible stopping pair (cf. Proposition 2). To do that, we delete from AD the longest word v_1 of such a pair. By Proposition 2 this word does not contribute to erasing letters in text t during the compression because there is nothing to erase after the last letter.

Hence we suppose that in our antidictionary AD this word is not included, or, equivalently, that the branch of trie $\mathcal{T}$ that has this word as unique leaf is pruned. In other words, we suppose from now on that antidictionary AD (and obviously all its subsets) has no stopping pair.

Algorithm SELF-AUTOMATON uses the previous hypothesis to reconstruct AD from $\mathcal{T}'$. More precisely, its input is a trie $\mathcal{T}'$, self-compressed from trie $\mathcal{T}$, with its transition function δ'. Its output is the automaton $\mathcal{A}(AD)$, where AD is the antidictionary represented by trie $\mathcal{T}$. It is similar to algorithm L-AUTOMATON. Indeed it makes a breadth-first traversal on states of the trie $\mathcal{T}$. It is possible to do this because, any time a state is reached, if a child was "erased" during the execution of SELF-COMPRESS, it is now created and added to the queue $\mathcal{Q}$. In order to create a new child, function δ must be previously restored, as done in algorithm L-AUTOMATON, by using the failure function f. When a leaf is reached in the self-compressed trie, the new stop criterion tells us that there is nothing more to reconstruct in that branch.

Trie $\mathcal{T}$ can be obtained from the automaton $\mathcal{A}(AD)$, output of next algorithm, by using a linear time algorithm described in [10].

The current situation in the next algorithm is as follows: when a node p is popped from the queue, trie $\mathcal{T}$ has been decompressed up to the level of p in $\mathcal{T}$, $f(p)$ is defined and function δ is defined for all previous nodes, which includes nodes at previous level. After processing p, δ is also defined for p and the failure function f is defined on its children.

```
SELF-AUTOMATON (trie T′)
i′ ← root of T′;
Q ← ∅;
for each a ∈ A
    if δ′(i′, a) is defined
        δ(i′, a) ← δ′(i′, a);
        f(δ(i′, a)) ← i′;
        add δ(i′, a) to Q;
    else
        δ(i′, a) ← i′;
while Q ≠ ∅
    extract p from Q;
    if p is not a leaf
        if δ(f(p), a) is a leaf for a ∈ A
            create p1;
            for each b ∈ A
                if δ′(p, b) is defined
                    δ′(p1, b) ← δ′(p, b);
            δ(p, ¬a) ← p1;
            δ(p, a) ← δ(f(p), a));
            f(p1) ← δ(f(p), ¬a));
            add p1 to Q;
        else
            for each a ∈ A
                if δ′(p, a) is defined
                    δ(p, a) ← δ′(p, a));
                    f(δ(p, a)) ← δ(f(p), a));
                    add δ(p, a) to Q;
                else
                    δ(p, a) ← δ(f(p), a));
    else
        for each a ∈ A
            δ(p, a) ← p;
return (Q, A, i′, Q \ {leaves}, δ);
```

Since there is a bijection between leaves of $\mathcal{T}$ and leaves of $\mathcal{T}'$, we can associate with any leaf q' of $\mathcal{T}'$ the same value $c(q)$ of the corresponding leaf q in $\mathcal{T}$. This is the number of bits that the word $w(q)$ leads to erase during the compression of text t. Analogously, as in the previous subsection, we can define functions *gain* and *pruned gain* and, as a first step, we can run algorithm SIMPLE PRUNING on $\mathcal{T}'$. At the same time we prune corresponding subtrees in $\mathcal{T}$ and obtain a trie $\mathcal{T}_1$. Doing so, the modified trie $\mathcal{T}_1$ represents a subset of *AD*. As a second step, we can use again algorithm SELF-COMPRESS on $\mathcal{T}_1$ to get ${\mathcal{T}'}_1$. Note that ${\mathcal{T}'}_1$ can be different from the pruned trie $\mathcal{T}'$ because pruning subtrees can affect self-compression.

We can iterate the above two steps for a fixed number of times or until the trie stabilizes.

VII. CONCLUSION

We have described DCA, a text compression method that uses some "negative" information about the text, represented in terms of antidictionaries. The advantages of the scheme are:

- it is fast at decompressing data,
- it is fast at compressing data for fixed sources,
- it has a synchronization property in the case of finite antidictionaries, property that leads to efficient parallel compression and to search engines on compressed data.

In the previous sections we presented some static DCA schemes in which the text to be compressed needs to be scanned twice. Starting from these static schemes, several variations and improvements can be proposed. These variations are all based on clever combinations of two elements that can be introduced in our model:

- statistic considerations,
- dynamic approaches.

These are classical features that are often included in other data compression methods.

Statistical considerations are used in the construction of antidictionaries. If a forbidden word is responsible for "erasing" few bits of the text in the compression algorithm of Section II and if its "description" as an element of the antidictionary is "expensive" then the compression ratio improves if it is not included in the antidictionary. This idea has been partially exploited in previous section. On the contrary, one can introduce into the antidictionary a word that is not forbidden but that occurs very rarely in the text. In this case, the compression algorithm will produce some "errors" or "mistakes" in predicting the next letter. In order to have a lossless compression, encoder and decoder must be adapted to manage such errors. Typical errors occur in the case of antidictionaries built for fixed sources as well as in the dynamic approach.

Even with errors, assuming that they are rare with respect to the maximum length of words of the antidictionary, our compression scheme preserves the synchronization property of Theorem 3. The use of errors becomes necessary for some artificial strings like 1^m0 if one wants to use a static aproach. Without errors and with a static approach, the algorithms described in previous section are unable to compress such strings.

Antidictionaries for fixed sources have also an intrinsic interest. A compressor generator, or compressor compiler, can create, starting from words obtained from a source S, an antidictionary that can be used to compress all other words from the same source S. Error management is essential for this kind of application. Having a fixed antidictionary makes the compression fast because basic operations are just table lookups.

In the dynamic approach, we construct the antidictionary and encode the text at the same time. The antidictionary is constructed (also with statistical consideration) by considering the whole text previously scanned or just a part of it. The antidictionary can change at any step and the algorithmic rules for its construction must be synchronized between encoder and decoder.

File	original size (in bytes)	compressed size (in bytes)
bib	111261	35535
book1	768771	295966
book2	610856	214476
geo	102400	79633
news	377109	161004
obj1	21504	13094
obj2	246814	111295
paper1	53161	21058
paper2	382199	2282
pic	513216	70240
progc	39611	15736
progl	71646	20092
progp	49379	13988
trans	93695	22695

Fig. 3. Compression ratios on files of the Calgary Corpus.

We have realized prototypes of the compression and decompression algorithms. They also implement the dynamic version of the method. They have been tested on the Calgary Corpus (see Figure 3), and experiments show that we get compression ratios equivalent to those of most common compressors (such as pkzip for example).

We are considering several generalizations:
- Compressor schemes and implementations of antidictionaries on more general alphabets or on other types of data (images, sounds, etc.),
- Use of lossy compression especially to deal with images,
- Combination of DCA with other compression schemes; for instance, using both dictionaries and antidictionaries like positive and negative sets of examples as in Learning Theory,
- Design of chips dedicated to fixed sources.

Several problems concerning the data compression scheme are still open. We list some of them.
- Are balanced sources dense inside the family of Markov sources? A positive answer would raise the question of adapting the scheme so that it becomes universal for Markov or ergodic sources. Can self compression be used to settle this question?
- Are there efficient algorithms to build good antidictionaries for syntactic sources, generated for instance by grammars? This raises a question of coding on a binary alphabet.
- What is the average of the maximum length of minimal forbidden words in texts of length n generated by an ergodic source having entropy H?
- How many times on the average should pruning and self compressing be iterated before the process stabilizes (see previous section)? We would expect a maximum of $\log n$ steps. Is the stabilized trie optimal?

Acknowledgments

We thanks M.P. Béal, M. Cohn, F.M. Dekking, R. Grossi and J. A. Storer for useful discussions and suggestions.

References

[1] J. G. Cleary T. C. Bell and I. H. Witten, *Text Compression*, Prentice Hall, 1990.

[2] J. Gailly, "Frequently asked questions in data compression," 2000, FAQ, URL http://www.faqs.org/faqs/faqs/compression-faq/.

[3] J. Gailly M. Nelson, *The Data Compression Book*, M&T Books, New York, NY, 1996.

[4] J. A. Storer, *Data Compression: Methods and Theory*, Computer Science Press, 1988.

[5] T. C. Bell I. H. Witten, A. Moffat, *Managing Gigabytes*, Van Nostrand Reinhold, 1994.

[6] C. Shannon, "Prediction and entropy of printed english," *Bell System Technical J.*, vol. January, 1951.

[7] A. Restivo M.-P. Béal, F. Mignosi, "Minimal forbidden words and symbolic dynamics," in *STACS'96*, C. Puech and R. Reischuk, Eds., number 1046 in Lecture Notes in Computer Science, pp. 555–566. Springer-Verlag, Berlin, 1996.

[8] A. Restivo M.-P. Béal, F. Mignosi and M. Sciortino, "Minimal forbidden words and symbolic dynamics," *Advances in Appl. Math.*, vol. To appear.

[9] A. Restivo M. Crochemore, F. Mignosi, "Minimal forbidden words and factor automata," in *MFCS'98*, J. Gruska L. Brim and J. Slatuška, Eds., number 1450 in Lecture Notes in Computer Science, pp. 665–673. Springer-Verlag, Berlin, 1998.

[10] M. Crochemore, F. Mignosi, and A. Restivo, "Automata and forbidden words," *Inf. Process. Lett.*, vol. 67, no. 3, pp. 111–117, 1998.

[11] A. Restivo M. Crochemore, F. Mignosi and S. Salemi, "Text compression using antidictionaries," in *ICALP'99*, J. Gruska L. Brim and J. Slatuška, Eds., number 1664 in Lecture Notes in Computer Science. Springer-Verlag, Berlin, 1999.

[12] C. Choffrut and K. Culik, "On extendibility of unavoidable sets," *Discrete Appl. Math.*, vol. 9, pp. 125–137, 1984.

[13] A. V. Aho and M. J. Corasick, "Efficient string matching: an aid to bibliographic search," *Commun. ACM*, vol. 18, no. 6, pp. 333–340, 1975.

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

[15] V. Diekert and Y. Kobayashi, "Some identities related to automata, determinants, and möbius functions," Report 1997/05, Universität Stuttgart, 1997.

[16] J. Berstel and D. Perrin, "Finite and infinite words," in *Algebraic Combinatorics on Words*, D. Perrin J. Berstel, Ed. Cambridge University Press, To appear.

[17] Y. Shibata, M. Takeda, A. Shinohara, and S. Arikawa, "Pattern matching in text compressed by using antidictionaries," in *CPM'99*, M. Crochemore and M. Paterson, Eds. 1999, number 1645 in Lecture Notes in Computer Science, pp. 37–49, Springer-Verlag, Berlin.

[18] M. P. Béal, *Codage Symbolique*, Masson, 1993.

[19] R. Ash., *Information Theory*, Tracts in mathematics. Interscience Publishers, J. Wiley & Sons, 1985.

[20] R. G. Gallager, *Information Theory and Reliable Communication*, J. Wiley and Sons, Inc., 1968.

[21] R. G. Gallager, *Discrete Stochastic Processes*, Kluver Acd. Publ., 1995.

[22] J. L. Snell J. G. Kemeny, *Finite Markov Chains*, Van Nostrand Reinhold, 1960.

[23] R. S. Ellis, *Entropy, Large Deviations, and Statistical Mechanics*, Springer Verlag, 1985.

[24] C. Hancart M. Crochemore, "Automata for matching patterns," in *Handbook of Formal Languages, Volume 2, Linear Modeling: Background and Application*, A. Salomaa G. Rozenberg, Ed. Springer-Verlag, 1997.

[25] R. Krichevsky., *Universal Compression and Retrieval*, Kluver Academic Publishers, 1994.

[26] M. Crochemore and R. Vérin, "On compact directed acyclic word graphs," in *Structures in Logic and Computer Science*, G. Rozenberg J. Mycielski and A. Salomaa, Eds., number 1261 in Lecture Notes in Computer Science, pp. 192–211. Springer-Verlag, Berlin, 1997.

Maxime Crochemore is professor and head of the Computer Science Research Laboratory at the University of Marne-la-Vallée (close to Paris). His research interests are algorithms on textual data, including pattern matching problems and data compression, and computational biology. He has co-authored the book "Text Algorithms" and is a steering committee member of the annual conference "Combinatorial Pattern Matching".

Filippo Mignosi is professor at the Department of Mathematics and Applications of the University of Palermo. His research interests include combinatorics on words and on more general informational structures, formal languages, automata theory, algorithms and data compression. He is visiting Brandeis University with a N.A.T.O. fellowship.

Antonio Restivo is professor at the Department of Mathematics and Applications of the University of Palermo. He is the chair of the board of mathematical and computer science studies, Faculty of Science. His research interests include automata theory, formal languages, combinatorics on words, coding theory, algorithms and data compression. He is the scientific national coordinator of the research project "Modelli innovativi di calcolo: metodi sintattici e combinatori", MURST, Italy.

Sergio Salemi is research fellow at the Department of Mathematics and Applications of the University of Palermo. His research interests include automata theory, combinatorics on words, coding theory, programming languages, algorithms and data compression.