Data Compression using Antidictionaries*

M. Crochemore[†], F. Mignosi[‡], A. Restivo[§], S. Salemi[¶]

March 20, 2000

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. Our algorithms can also be presented as "compilers" that create compressors dedicated to any previously fixed source. 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.

1 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. Contrary 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. [8], [17], [23], [25] and [26]), our method takes advantage from words that do not occur as factor in the text, *i.e.*, that are forbidden. Such sets of words are called here antidictionaries.

*DCA home page at URL `http://www-igm.univ-mlv.fr/~mac/DCA.html`

†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, 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.

We describe a static compression scheme that runs in linear time (Sections 2 and 3) including the construction of antidictionaries (Section 5). Variations using statistical or dynamical considerations are discussed in the conclusion (Section 7)

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 admits as suffix a word u' such that $u = u'x \in AD$ with $x \in \{0,1\}$, *i.e.*, u is forbidden, then surely the letter following v in the text cannot be x and, since the alphabet is binary, it is the letter $y \neq x$. In other terms, we know in advance the next letter y, 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 [24] 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 4) 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. [4] and [5]).

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 $u = u'x$, $x \in \{0,1\}$, in AD such that u' is a suffix of v? In the case of positive answer the output should also include the letter y defined by $y \neq x$. 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 details in [10] and [11].

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 2 we give the definition of Forbidden Words and of antidictionaries. We describe DCA, our text compression and decompression algorithms (binary oriented) assuming that the antidictionary is given. In Section 3 we describe a data structure for finite antidictionaries that allows to answer in efficient way to the queries needed by our compression and decompression algorithms; we show how to implement it given a finite antidictionary. The compression is also described in terms of transducers, which is valid only in the case of rational antidictionaries. We end the section by proving the synchronization property. In Section 4 we evaluate the compression rate of our compression algorithm relative to a given antidictionary. In Section 5 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. We discuss improvements and generalizations in Section 7.

Some of the results present in this paper have been synthetically stated in [12].

2 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 7.

Let w be a finite binary word and let $F(w)$ be the set of factors of w. For instance, if $w = \mathtt{01001010}$ then $F(w) = \{\varepsilon, \mathtt{0}, \mathtt{1}, \mathtt{00}, \mathtt{01}, \mathtt{10}, \mathtt{001}, \mathtt{010}, \mathtt{100}, \mathtt{101}, \ldots\}$ 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*. The set of such words AD is called an *antidictionary* for the language $F(w)$. Antidictionaries can be finite as well infinite. For instance, if $w = \mathtt{01001010}$ the words $\mathtt{11}$, $\mathtt{000}$, and $\mathtt{10101}$ are forbidden and the set $\{\mathtt{11}, \mathtt{000}, \mathtt{10101}\}$ is an antidictionary for $F(w)$. For instance, if $w_1 = 001001001001$ the infinite set of all words that have two digit 1 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 one word $u = u'x$, $x \in \{\mathtt{0}, \mathtt{1}\}$, in the antidictionary AD such that u' is a suffix of v, then surely the letter following v cannot be x, *i.e.*, the next letter is y, $y \neq x$. In other words, we know in advance the next letter y 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 (anti-dictionary *AD*, word $w \in \{0,1\}^*$)
1. $v \leftarrow \varepsilon$; $\gamma \leftarrow \varepsilon$;
2. **for** $a \leftarrow$ first to last letter of w
3. **if** for every suffix u' of v, $u'0, u'1 \notin AD$
4. $\gamma \leftarrow \gamma.a$;
5. $v \leftarrow v.a$;
6. **return** $(|v|, \gamma)$;

As an example, let us run the algorithm ENCODER on the string $w = 01001010$ with the antidictionary $AD = \{000, 10101, 11\}$. 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	$\gamma(w) = \varepsilon$	
$v_1 = 0$	$\gamma(w) = 0$	
$v_2 = 01$	$\gamma(w) = 01$	$u = 11 \in AD$
$v_3 = 010$	$\gamma(w) = 01$	
$v_4 = 0100$	$\gamma(w) = 010$	$u = 000 \in AD$
$v_5 = 01001$	$\gamma(w) = 010$	$u = 11 \in AD$
$v_6 = 010010$	$\gamma(w) = 010$	
$v_7 = 0100101$	$\gamma(w) = 0101$	$u = 11 \in AD$
$v_8 = 01001010$	$\gamma(w) = 0101$	$u = 10101 \in AD$
$v_9 = 010010100$	$\gamma(w) = 0101$	$u = 000 \in AD$
$v_{10} = 0100101001$	$\gamma(w) = 0101$	$u = 11 \in AD$

Remark that the function γ is not injective.

For instance $\gamma(010010100) = \gamma(01001010001) = 0101$.

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 $u = u'x$, $x \in \{0,1\}$, in the antidictionary *AD* such that u' is a suffix of v, then, the output letter is y, $y \neq x$. Otherwise, the next letter is read from the input γ.

Decoder (anti-dictionary *AD*, word $\gamma \in \{0,1\}^*$, integer n)

1. $v \leftarrow \varepsilon$;
2. **while** $|v| < n$
3.   **if** for some u' suffix of v and $x \in \{0,1\}$, $u'x$ belongs to *AD*
4. $v \leftarrow v \cdot \neg x$;
5.   **else**
6. $b \leftarrow$ next letter of γ;
7. $v \leftarrow v \cdot b$;
8. **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 $u = u'x$, $x \in \{0,1\}$, in *AD* such that u' is a suffix of v? In case of a positive answer the output should also include the letter y defined by $y \neq x$. Notice that the letter x 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. Light variations of 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 have 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 [24] 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 $\mathtt{01\bar{0}0\bar{1}\bar{0}1\bar{0}\bar{0}\bar{1}\bar{0}}$, where bars indicate the letters erased in the compression algorithm.

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 $u = u'x$, $x \in \{0,1\}$ 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.

3 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 [10] and [11]. The same construction has been discovered by Choffrut *et al.* [7], it is similar to a description given by Aho-Corasick ([1], see [13]), by Diekert *et al.* [15], and it is related to a more general construction given in [6].

The wanted 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: $MF(L) = AL \cap LA \cap (A^* \setminus L)$. After the note made at the end of the previous section, from now on in the paper, we consider only antidictionaries that are anti-factorial languages.

The following theorem is proved in [11]. It is based on an algorithm that has AD as (finite) input, either in the form of a list of words or in the form of a trie $\mathcal{T}$ representing the set. 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 built the automaton associated with the anti-factorial language corresponding to any set of words.

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

We report here, for sake of completeness, the algorithm described in [11] that, given the trie $\mathcal{T}$ of an anti-factorial language, builds in linear time the automaton $\mathcal{A}(AD)$.

The input is the trie $\mathcal{T}$ that represents AD. It is a tree-like automaton accepting the set AD and, as such, it is noted (Q, A, i, T, δ'). 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 M. For a state au ($a \in A$, $u \in A^*$), $f(au)$ is $\delta'(i, u)$, quantity that may happen to be u itself. Note that $f(i)$ is undefined, which justifies a specific treatment of the initial state in the algorithm.

L-AUTOMATON (trie $\mathcal{T} = (Q, A, i, T, \delta')$)
for each $a \in A$
    if $\delta'(i, a)$ defined
        $\delta(i, a) \leftarrow \delta'(i, a)$;
        $f(\delta(i, a)) \leftarrow i$;
    else
        $\delta(i, a) \leftarrow i$;
for each state $p \in Q \setminus \{i\}$ in width-first search **and** each $a \in A$
    if $\delta'(p, a)$ defined
        $\delta(p, a) \leftarrow \delta'(p, a)$;
        $f(\delta(p, a)) \leftarrow \delta(f(p), a)$;
    else if $p \notin T$
        $\delta(p, a) \leftarrow \delta(f(p), a)$;
    else
        $\delta(p, a) \leftarrow p$;
return $(Q, A, i, Q \setminus T, \delta)$;

3.1 Example

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

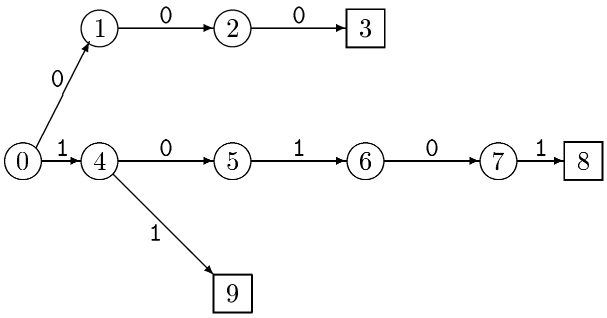

Figure 1: Trie of the factor code $\{000, 10101, 11\}$. Squares represent terminal states.

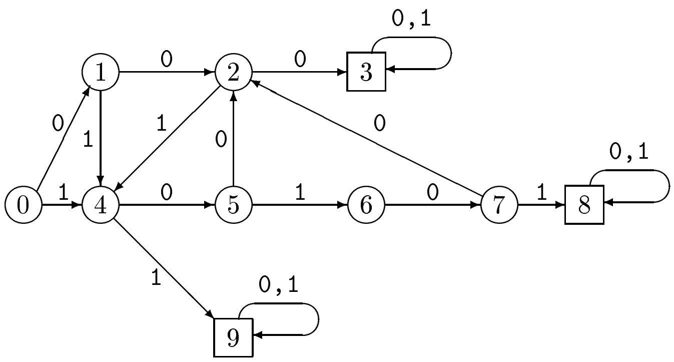

Figure 2: Automaton accepting the words that avoid the set $\{\texttt{000}, \texttt{10101}, \texttt{11}\}$. Squares represent non-terminal states (sink states).

3.2 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).*

As to concern the algorithm DECODER, remark (see Section 2) 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 run of the decoding algorithm realized by $B'(AD)$ in any state $q \in p(q_{m'})$, and, for different states, we obtain different decoding. So, we need a 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 [21]).

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 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 2 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.

3.3 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* as described before 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. $\diamondsuit$

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 [3]).

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 2 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.

4 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 defines naturally 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 that recognizes 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 the 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. [2]) 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. [2]). 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. [2], [19], [20] and [18]). Recall (cf. Theorem 6.4.2 of [2]) 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 an 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. $\diamondsuit$

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 $\mathcal{A}(AD)$. $\diamondsuit$

Through a well-known results on "large deviations" (cf. Problem IX.6.7 of [16]), 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 $X_m(\epsilon) = \{ \mathbf{q} \mid \mathbf{q}$ has m 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 $L_{m,j}(\mathbf{q}) > \mu_j + \epsilon/d$, where d is the cardinality of the set 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. $\diamondsuit$

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.*

5 How to build Antidictionaries

In practical applications the antidictionary could be not *a priori* given but it must be derived either from the text to be compressed or from a family of texts belonging to the same source to which the text to be compressed is supposed to belong.

There exist several criteria to build efficient antidictionaries that depend 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 [11]). 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 according to 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, and it 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, statistic considerations and the possibility of doing "errors" in predicting the next letter play an important role, as discussed in Section 7.

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) than 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 $w \in \{0,1\}^*$, integer $k > 0$)
1. $i \leftarrow$ new state; $Q \leftarrow \{i\}$;
2. $level(i) \leftarrow 0$;
3. $p \leftarrow i$;
4. **while not** end of string w
5. $a \leftarrow$ next letter of w;
6. $p \leftarrow$ Next(p, a, k);
7. **return** trie (Q, i, Q, δ), function f;

Next (state p, letter a, integer $k > 0$)
1. **if** $\delta(p, a)$ defined
2. **return** $\delta(p, a)$;
3. **else if** $level(p) = k$
4. **return** Next$(f(p), a, k)$;
5. **else**
6. $q \leftarrow$ new state; $Q \leftarrow Q \cup \{q\}$;
7. $level(q) \leftarrow level(p) + 1$;
8. $\delta(p, a) \leftarrow q$;
9. **if** $(p = i)$ $f(q) \leftarrow i$;
10. **else** $f(q) \leftarrow$ Next$(f(p), a, k)$;
11. **return** q;

Build-AD (trie (Q, i, Q, δ), function f, integer $k > 0$)
1. $T \leftarrow \emptyset$; $\delta' \leftarrow \delta$;
2. **for** each $p \in Q$, $0 < level(p) < k$, in width-first order
3. **for** $a \leftarrow 0$ then 1
4.   **if** $\delta(p, a)$ is undefined **and** $\delta(f(p), a)$ is defined
5. $q \leftarrow$ new state; $T \leftarrow T \cup \{q\}$;
6. $\delta'(p, a) \leftarrow q$;
7. $Q \leftarrow Q \setminus \{$states of Q from which no δ'-path leads to $T\}$
8. **return** trie $(Q \cup T, i, T, \delta')$;

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 [9] 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 twice the text: 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 2, 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 2. 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 to 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 4, approximates the source of the text, then, by the results of Section 4, 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 by 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. [22]).

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.

6 Pruning Antidictionaries

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

In this section, however, we suppose to have enough resources to build, in linear time, a suffix or a factor automaton (or their compacted version, cf. [14]) of the finite text string to be compressed. From these structure we can obtain in linear time a representation of *all* minimal forbidden words of the text, *i.e.* a trie of the antidictionary of all minimal forbidden words of the text (cf. [11]). 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 use the fact 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.

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, which produces higher compression ratios.

6.1 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 [11]. 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 3, the states of the trie $\mathcal{T}$ to the prefixes of the minimal forbidden words, then the function w is the identity.

We define a function $c(q)$ that associates with any leaf q of $\mathcal{T}$ the number of bits 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. In another words, the number $c(q)$ is the number of times that a state p in the automaton $\mathcal{A}(AD)$ such that $\delta(p, a) = q$ is encountered while reading the text t from the first to the last but one letter (cf. Section 3 and Theorem 1). Indeed there is nothing to erase after the last letter. By Theorem 1, the function $c(q)$ can be computed in linear time.

We can now give the main definition of this subsection. The *gain* (saving) of a subtree S of the entire anti-dictionary T is $g(S) = \Sigma(c(q) \mid q \text{ leaf of } S) - 2m_S$ where m_S is the number of nodes of S.

It is not difficult to see that we can give the following equivalent definition for $g(S)$.

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

where $M = \max(g(S_1), g(S_2), g(S_1) + g(S_2)) - 2$.

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 $\mathcal{T}$.

Indeed the number of bits that have to be sent after compression is $2 \log n$ to encode the length n of the text t, plus a description of the antidictionary $\mathcal{T}$, plus $\gamma(t)$ the text compressed using $\mathcal{T}$, *i.e.*

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

Since $2 \log n + n$ is fixed and since the gain $g(\mathcal{T})$ is the sum of the gain of its subtrees, then pruning subtrees of $\mathcal{T}$ that have a negative gain increases the gain of $\mathcal{T}$ and, consequently, decreases the number of bits that have to be send after compression. Therefore we can describe the following algorithm.

SIMPLE PRUNING (trie $\mathcal{T}$, function gain g)
1. eliminate subtrees S of $\mathcal{T}$
   for which $g(S) \leq 0$;
2. **return** modified trie $\mathcal{T}$;

From the descriptions above, we have the next simple results.

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

Proposition 2 *If trie $\mathcal{T}$ represents an antifactorial language AD then the trie output by algorithm* SIMPLE PRUNING *represents a subset of AD.*

6.2 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 $AD \setminus \{v\}$. Let us denote by $\gamma_1(v)$ the compressed version of v by using $AD \setminus \{v\}$. We can think to send, in a static approach, the set $X_1 = \{\gamma_1(v) \mid v \in AD\}$ instead of AD itself in order to get a better compression.

But the set X_1 is not necessarily any more an antifactorial language, nor even a prefix code.

The reader can easily verify that if $AD = \{\texttt{11, 000, 10101, 00100100, 1010010100101}\}$ then $X_1 = \{\texttt{11, 000, 111, 0000, 1111,}\}$. Also, if $AD = \{\texttt{10, 110}, \cdots, \texttt{1}^{\texttt{n}}\texttt{0}\}$ then, for any $n \geq 0$ then $X_1 = \{10\}$.

Consequently, the knowledge alone of the trie of set X_1 is not sufficient to reconstruct *AD*, but some further information and more complex algorithms will be needed for this purpose.

However, a different approach that makes use of the same idea 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*.

Given a word $v \in AD$ we compress it using antidictionary AD' that dynamically changes at any step of the **while** loop at line 2 of algorithm ENCODER. If a prefix u of v is read, antidictionary AD' is composed by all words belonging to AD that have length strictly smaller than the length $|u|$ of prefix u. Let us call $\gamma_2(v)$ the compressed version of v by using AD' and $X_2 = \{\gamma_2(v) \mid v \in AD\}$
.

As an immediate consequence of previous definitions we have the following result, that is used to recover $\mathcal{T}$ from $\mathcal{T}'$.

Proposition 3 *If for any $v = ua, v_1 = u_1 b \in AD$, with $a, b \in \{0,1\}$, $a \neq b$, and $|v_1| \geq |v|$ we have that u is not a suffix of u_1, then for any $v \in AD$, the last letter of v is not erased during the compression to get $\gamma_2(v)$.*

This kind of self-compression can be performed in linear time by next algorithm SELF-COMPRESS that has as input the trie $\mathcal{T}$ that represents *AD* and the function δ of automaton $\mathcal{A}(AD)$ (cf. algorithm L-AUTOMATON). It uses a width-first search on the nodes of $\mathcal{T}$ implemented with queue $\mathcal{Q}$ and, at the same time, it creates a self-compressed version $\mathcal{T}'$ of $\mathcal{T}$ that represents the set X_2. Its output is $\mathcal{T}'$. States i and i' are the initial states of $\mathcal{T}$ and $\mathcal{T}'$ respectively.

SELF-COMPRESS (trie $\mathcal{T}$, function δ))
$\mathcal{Q} \leftarrow \{(i, i')\}$;
while $\mathcal{Q} \neq \emptyset$
  extract (p, p') from $\mathcal{Q}$;
  if q_0 and q_1 children of p
    create q_0' and q_1' as children of p';
    add (q_0, q_0') and (q_1, q_1') to $\mathcal{Q}$;
  else if $q = \delta(p, a)$, $a \in A$
    if $\delta(p, a)$ is a leaf
      add (q, p') to $\mathcal{Q}$;
    else create q' as a-child of p';
      add (q, q') to $\mathcal{Q}$;
return trie $\mathcal{T}' = (Q', A, i', T, \delta'_{\mathcal{T}'})$;

Remark that the function $\delta'_{\mathcal{T}'}$ is implicitly defined when, at lines 5 or 10, children of p' are created.

Notice also that, whenever a node $p \in \mathcal{T}$ has two children then the same holds for its corresponding node $p' \in \mathcal{T}'$ (cf. lines 4 and 5 of previous algorithm). Therefore, $\mathcal{T}$ and $\mathcal{T}'$ have the same number of internal nodes that have two children and, consequently, the same number of leaves. Hence, the set X_2 that $\mathcal{T}'$ represents, is a prefix code.

Next algorithm has as input a self compressed trie $\mathcal{T}'$. Its output is the automaton $\mathcal{A}(AD)$, where AD is the antidictionary represented by trie $\mathcal{T}$. We suppose that AD satisfy the hypothesis of Proposition 3.

It is similar to algorithm L-AUTOMATON. Indeed it makes a width-first search on the states of the non self-compressed trie $\mathcal{T}$. It is possible to do this because at any time a state is reached, if a child was "erased" during the SELF-COMPRESS algorithm, 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.

Trie $\mathcal{T}$ can be obtained from the output automaton by using a linear time algorithm described in [11].

```
SELF-AUTOMATON (trie T' = (Q', A, i', T, δ'_T'))
Q ← ∅;
for each a ∈ A
    if δ'_T'(i', a) is defined
        δ(i', a) ← δ'_T'(i', a);
        f(δ(i', a)) ← i';
        add delta(i', a) to Q;
    else
        δ(i', a) ← i';
while Q ≠ ∅
    p' ← DEQUEUE Q;
    if δ'_T'(p, 0) and δ'_T'(p, 1) are defined
        for each a ∈ A
            δ(p, a) ← δ'_T'(p, a);
            f(δ(p, a)) ← δ(f(p), a);
            add delta(p', a) to Q;
    else if p ∉ T
        if δ(f(p), a) ∈ T create p1;
            f(p1) ← f(p),
            p̂ ← father of p
            if p is x-child of p̂
                p1 ← x-child of p̂;
            p ← ¬a-child of p1 ;
            δ(p1, a) ← δ(f(p), a);
            f(p) ← δ(f(p), ¬a);
        else
            δ(p, a) ← p;
return (Q', A, i', Q \ T, δ);
```

We now prove the correctness of algorithm SELF-AUTOMATON.

We first notice that if AD is an antifactorial antidictionary of a text t then if $v = ua, v_1 = u_1 b \in AD$, with $a, b \in \{0, 1\}$, $a \neq b$, and u is a suffix of u_1, then u_1 must be a suffix of t and it must not appear elsewhere in t. This is because, otherwise, since u is a suffix of u_1, the next position must be not letter a (because ua is forbidden) and must be not letter b (because $u_1 b$ is forbidden).

Therefore, among all couples (v, v_1) of minimal forbidden words of a single text t only one can satisfy the previous hypothesis. Moreover, the longest word of the couple 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, the branch of trie $\mathcal{T}$ that has this word as unique leave is pruned. This can easily be done in time proportional to $|t|$ by using automaton $\mathcal{A}(AD)$).

Hence we can suppose that antidictionary AD (and obviously all its subsets) satisfies the hypothesis of Proposition 3.

As a second point we notice that if AD satisfy the hypothesis of Proposition 3, then the last letter of any word $v \in AD$ is not erased by algorithm SELF-COMPRESS. This means that, in this case, when we use algorithm DECODER to recover v from $\gamma_2(v)$, we do not need to know *a priori* the length of v, because we can use a new "stop" criterion (cf. Section 2): stop decoding after that the last letter of $\gamma_2(v)$ has been read. And this is the stop criterion used in algorithm SELF-AUTOMATON to univoquely reconstruct $\mathcal{T}$ from $\mathcal{T}'$.

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 $\mathcal{T}$, that is the number of bits that the word $w(q)$ allows to erase during the compression of the text t. Analogously, as in the previous subsection, we can define a function gain and, as first step we can run algorithm SIMPLE PRUNING on $\mathcal{T}'$. At the same time we prune subtrees in $\mathcal{T}$ that correspond to the subtrees pruned in $\mathcal{T}'$.

The modified trie $\mathcal{T}$ represents a subset of AD. As a second step, we can use algorithm SELF-COMPRESS on modified $\mathcal{T}$ and obtain a modified $\mathcal{T}'$ (that could be different from the pruned one).

We can iteratively repeat the above two steps for a chosen number of times or until trie $\mathcal{T}$ stabilizes.

7 Conclusion

In the previous sections we presented static compression schemes in which we need to read twice the text. Starting from the 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:

a) statistic considerations,

b) dynamic approaches.

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

Statistic considerations are used in the construction of antidictionaries. If a forbidden word is responsible of "erasing" few bits of the text in the compression algorithm of Section 2 and its "description" as an element of the antidictionary is "expensive" then the compression rate improves if it is not included in the antidictionary. On the contrary, one can introduce in 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 longest word (length) of the antidictionary, our compression scheme preserves the synchronization property of Theorem 3.

Antidictionaries for fixed sources have also an intrinsic interest. A compressor generator, or compressor compiler, can create, starting from words obtained by a source S an antidictionary that can be used to compress all the other words from the same source S, taking into account possible "errors" in predicting the next letter.

In the dynamic approach we construct the antidictionary and we encode the text at the same time. The antidictionary is constructed (also with statistical consideration) by considering the part of text previously read. The antidictionary can change at any step and the algorithmic rules for its construction must be known in advance by both encoder and decoder.

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 next table), and experiments show that we get compression ratios equivalent to those of most common compressors (such as pkzip for example).

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

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

- it is fast at decompressing data,
- it produces compressor generators (compressor compilers) for fixed sources,
- 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.

We are considering several generalizations:

- compressor schemes and implementations of antidictionaries on more general alphabets (including, for instance, arithmetic coding),
- 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.

Acknowledgments

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

References

[1] A. V. Aho, M. J. Corasick. Efficient string matching: an aid to bibliographic search. *Comm. ACM* 18:6 (1975) 333–340.

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

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

[4] M.-P. Béal, F. Mignosi, A. Restivo. Minimal Forbidden Words and Symbolic Dynamics. in (*STACS'96*, C. Puech and R. Reischuk, eds., LNCS 1046, Springer, 1996) 555–566.

[5] M.-P. Béal, F. Mignosi, A. Restivo, M. Sciortino. Minimal Forbidden Words and Symbolic Dynamics. To appear in Advances in Appl. Math.

[6] J. Berstel, D. Perrin. Finite and infinite words. in (*Algebraic Combinatorics on Words*, J. Berstel, D. Perrin, eds., Cambridge University Press, to appear) Chapter 1. Available at URL `http://www-igm.univ-mlv.fr/~berstel`

[7] C. Choffrut, K. Culik. On Extendibility of Unavoidable Sets. Discrete Appl. Math., **9**, 1984, 125-137.

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

[9] M. Crochemore, C. Hancart. Automata for matching patterns. in (*Handbook of Formal Languages*, G. Rozenberg, A. Salomaa, eds.", Springer-Verlag", 1997, Volume 2, *Linear Modeling: Background and Application*) Chapter 9, 399–462.

[10] M. Crochemore, F. Mignosi, A. Restivo. Minimal Forbidden Words and Factor Automata. in (*MFCS'98*, L. Brim, J. Gruska, J. Slatuška, eds., LNCS 1450, Springer, 1998) 665–673.

[11] M. Crochemore, F. Mignosi, A. Restivo. Automata and Forbidden Words. *Information Processing Letters* 67 (1998) 111–117.

[12] M. Crochemore, F. Mignosi, A. Restivo, S. Salemi. Text Compression using Antidictionaries. in (*ICALP'99*, L. Brim, J. Gruska, J. Slatuška, eds., LNCS 1664, Springer, 1999).

[13] M. Crochemore, W. Rytter. *Text Algorithms.* Oxford University Press, 1994.

[14] M. Crochemore, R. Vérin. On compact directed acyclic word graphs. In J. Mycielski, G. Rozenberg, and A. Salomaa, editors, *Structures in Logic and Computer Science*, LNCS 1261, Springer-Verlag, 192–211, 1997.

[15] V. Diekert, Y. Kobayashi. *Some Identities Related to Automata, Determinants, and Möbius Functions.* Report Nr. 1997/05, Universität Stuttgart, Fakultät Informatik, 1997.

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

[17] J. Gailly. *Frequently Asked Questions in data compression*, Internet. URL `http://www.faqs.org/faqs/faqs/compression-faq/`

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

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

[20] Robert G. Gallager *Discrete Stochastic Processes.* Kluver Acd. Publ., 1995.

[21] Y. Shibata, M. Takeda, A. Shinohara, S. Arikawa. *Pattern matching in text compressed by using antidictionaries* Proc. 10th Ann. Symp. on Combinatorial Pattern Matching, LNCS 1645, Springer-Verlag, 37-50, 1999.

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

[23] M. Nelson, J. Gailly. *The Data Compression Book.* M&T Books, New York, NY, 1996. 2nd edition.

[24] C. Shannon. Prediction and entropy of printed english. *Bell System Technical J.*, 50-64, January, 1951.

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

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