1999

Off-Line Compression by Greedy Textual Substitution

Alberto Apostolico

Stefano Lonardi

Report Number:
99-043

OFF-LINE COMPRESSION BY GREEDY TEXTUAL SUBSTITUTION

Alberto Apostolico
Stefano Lonardi

Department of Computer Science
Purdue University
West Lafayette, IN 47907

CSD TR #99-043
December 1999

Off-line Compression by Greedy Textual Substitution

Alberto Apostolico Stefano Lonardi
Purdue University and Università di Padova

Abstract— **Greedy off-line textual substitution refers to the following approach to compression or structural inference. Given a long textstring x, a substring w is identified such that replacing all instances of w in x except one by a suitable pair of pointers yields the highest possible contraction of x; the process is then repeated on the contracted textstring, until substrings capable of producing contractions can no longer be found. The paper examines computational issues arising in the implementation of this paradigm and describes some applications and experiments.**

Keywords: off-line textual substitution, dynamic text compression, compression of biological sequences, grammatical inference, substring statistics, augmented suffix tree.

I. Introduction

In data compression by textual substitution (see, e.g., [1], [2], [3]), substrings with multiple occurrences in a textstring are replaced by a suitable set of pointers to a unique common copy (for instance, by giving (1) a textstring position starting from which the substring can be recopied, and (2) the length of that substring). Disparate conventions, regarding issues such as the location of the common copy, and the mechanics of the encoding-decoding process, give rise to various *macro schemes* of compression. In general, the relative performance of such schemes depends on many factors, including the often subtle interplay between pointer sizes and dictionary parameters (say, number of entries, and average length). Partly in response to this fact, techniques were devised for the compact encoding of integers in an unbounded domain (see, e.g., [4], [5], [7]). Unfortunately, however, the optimal implementation of the majority of macro schemes translates into $\mathcal{NP}$-complete problems [3], even before the problem of encoding of pointers is taken into account. One noteworthy exception to this rule is represented by the well-known Lempel-Ziv schemes [8], [9], [10], which attain asymptotic optimality both in terms of compression achieved and algorithmic complexity. In Lempel-Ziv data compression (LZ), data can be processed *on-line* as it is read, a feature that nicely fits the standard paradigm of sequential transmission. The uni-directional or "polar" nature of pointers is crucial in determining the computational efficiency inherent to this scheme.

In some applications, like for instance in the production of a CD-ROM or magnetic disk for massive data dissemination, one could afford to perform the compression *off-line*, in particular, to issue pointers in either direction if this brings an increase in compression. Off-line heuristics may be expected to introduce extra time by whatever sequential implementation, but their possible implementation on parallel, perhaps dedicated architectures (see, e.g., [11], [12]), may be expected to achieve sufficient speed to process streams of large consecutive textfile windows consecutively in real-time for any practical purpose. Within the realm of sequential computation, investing more time in the compression may be desirable and feasible for information destined to be massively distributed, as long as the decompression can be still carried out fast and on-line [13]. In other situations, such as e.g., in backup archiving, the odds of having to restore the data might be feeble enough that even the requirement that this phase be on-line could be forfeited. Finally, as we briefly illustrate at the end of this paper, the study and implementation of macro schemes of the kind considered here may be of some interest in the germane field of inference of hierarchical structures or grammars for sequences (see, e.g., [14], [15], [17]).

The idea that some of the polarity or greediness inherent to LZ schemes could be traded in for increased compression is intuitively appealing and not new. In [18], [19], [20], for instance, the authors discuss variations such as, e.g., relaxing the longest-match criterion in determining the next phrase within an LZ parse. The underlying goal is to try and converge faster to the entropy of the source. In view of the intractability of optimal off-line macro schemes, we concentrate here on the implementation of approx-

Work supported in part by NSF Grants CCR-9201078 and CCR-9700276, by NATO Grant CRG 900293, by British Engineering and Physical Sciences Research Council Grant GR/L19362, by Purdue Research Foundation Grant 690-1398-3145, and by the Italian Ministry of University and Research. Preliminary results related to this paper were presented at the Annual Data Compression Conference of the IEEE, Snowbird, Utah, 1998.

Corresponding Address: Department of Computer Sciences, Purdue University, Computer Sciences Building, West Lafayette, IN 47907-1398, USA. {axa,stelo}@cs.purdue.edu.

imate methods such as one of the simplest possible *steepest descent* paradigm. This will consist of performing repeated stages in each one of which we identify a substring of the current version of the text yielding the maximum compression, and then replace all those occurrences except one with a pair of pointers to the untouched occurrence. This is somewhat dual with respect to the bottom up offline scheme introduced by Rubin [21] and recently revived by [22]. As we shall see, this simple scheme already poses some interesting algorithmic problems, some of which we discuss in detail. However, the main issue that we try to address here is that of whether and to what extent a greedy use of bi-directional pointers can yield good compression. As it turns out, the method does outperform all current Lempel-Ziv implementations in most of the cases. More interestingly, it performs quite well on biological sequences and sequence families, where it beats all other generic compression methods, and approcahes the performance of methods specifically built around some peculiar regularities of DNA sequences, such as tandem repeats and palindromes, that are neither distinguished nor treated selectively here. The most interesting performances, however, are obtained in the compression of entire groups of genetic sequences forming *families* with similar characteristics. This is becoming a standard and useful way to group sequences in a growing number of important specialized databases. On such inputs, the approach presented here yields scores that are not only better than those of any other method, but also improve increasingly with increasing input size. This is to be attributed to a certain ability to capture distant relationships among the sequences in a family, a feature the merits of which were dramatically exposed in the recent paper [43].

Biological sequences, specially DNA, have been long recognized among important classes of data for which the two tasks of compression and interpretation are often and subtly intertwined. (see, e.g., [23]). The deoxyribonucleic acid (DNA) constitutes the physical medium in which all properties of living organisms are encoded. The knowledge of its sequence is fundamental in molecular biology. Important molecular biology databases (e.g., EMBL, Genbank, DDJB, Entrez, SwissProt, etc.) have been developed to collect hundreds of thousand of sequences of nucleotides and amino-acids from biological laboratories all over the world. The size of these databases, that is currently in the order of thousands of gigabytes, grows at an exponential rate. DNA compression by standard methods such as, e.g., the Lempel-Ziv family of schemes does not seem to fully exploit the redundancies inherent to those sequences. The design of *ad hoc* methods for the compression of genetic sequences constitutes, therefore, an interesting and worthwhile task. Along these lines, a corpus of specialized approaches to DNA compression has been developed in the recent past. As highlighted above, pendant notions of information content and structure have been associated with the compressibility of a sequence. From such a perspective, the amount of compression achievable on genetic sequences has been used in the detection of fragments carrying biological significance, or in assessing the relatedness of fragments and sequences. We refer to, e.g., [24], [25], [23], [26], [27], [28], [29], [30], [31], [32], [33] and references therein for a sampler of the rich literature existing on these subjects.

The structure of the paper is as follows. In sections II and III we give some background and notation, and describe a data structure used to gather the statistics of the text. The overall design is presented in sections IV which is followed by a presentation of the main experimental results in section V. A discussion of finer implementation details and some final remarks conclude the paper.

II. Substring Statistics Without Overlap

We use Σ to denote an *alphabet* of *symbols*. For a *string* x over Σ, the number of consecutive symbols in x is the *length* $|x|$ of x, and we write $x[i]$, $1 \leq i \leq |x|$ to indicate the i-th symbol in x. In the following, we assume $|x| = n$. We use $x[i, j]$ shorthand for the *substring* w of x composed by $x[i] \cdot x[i+1] \cdot \ldots \cdot x[j]$ where $1 \leq i \leq j \leq |x|$, and $x[i, i] = x[i]$. Finally, substrings in the form $x[1, j]$ are called *prefixes* of x, and substrings in the form $x[i, |x|]$ are called *suffixes* of x. For any substring w of x, we denote by f_w the number of *nonoverlapping* occurrences of w in x. Clearly, f_w may be different from the total number of occurrences of w. For example, $w = \mathrm{aba}$ occurs 11 times in $x = \mathrm{abaababaabaababaababababaa}$, with starting positions in the set $\{1, 4, 6, 9, 12, 14, 17, 19, 21, 23, 25\}$ (cf. Fig. 1). However, occurrences starting at positions 4 and 6, or 12 and 14, etc., overlap with each other. We can have no more than 7 occurrences of w in x so that no two of them overlap. For instance, we could take those with starting positions in $\{1, 4, 9, 12, 17, 21, 25\}$. Thus, $f_{aba} = 7$. To understand our interest in the count of nonoverlapping occurrences, assume that a substring w appears repeatedly in x. Then, replacing all occurrences of w except one with a pointer to the unique reference copy might yield a more compact description of x. If f_w is known, then it is also possible to assess beforehand the contraction in length that x would undergo following such an encoding. If, now, we were asked

Fig. 1. Overlapping and non-overlapping occurrences

to identify the one substring w inducing the highest contraction on x, we could clearly do so based on the f-value and length of the individual substrings. Choosing instead on the basis of the total number of occurrences would neither guarantee nor allow us to pre-compute the best contraction.

The computation of the statistics of all substrings of a string x is an easy application of the *suffix tree* T_x of x. As is well known, the latter is a trie (digital search tree) collecting all the suffixes of $x\$$, where $\$$ is a special symbol not included in Σ. The tree in compact form is built by iterated insertion of consecutive suffixes in $\Theta(n^2)$ worst case time and $O(n \log n)$ expected time (see, e.g., [34]). A number of more clever constructions are available achieving linear time for finite alphabets (see, e.g., [35]). The number of occurrences (with overlap) of a string w of x is trivially given by the number of leaves reachable from the node closest to the locus of w in T_x, irrespective of whether or not w ends in the middle of an arc. Thus, labeling every internal node α of T_x with the number $c(\alpha)$ of the leaves in the subtree rooted at α yields this statistics for all substrings of x.

The problem becomes more involved if we wanted to build a similar index for the statistics without overlap. A perusal of Figure 2 shows that this transition induces a twofold change in our structure: on the one hand, the weight in each node does no longer necessarily coincide with the number of leaves; on the other, extra nodes must be now introduced to account for changes in the statistics that occur in the middle of arcs. The efficient construction of this augmented index in minimal form (i.e., with the minimum possible number of unary nodes) is quite elaborate [36]. For a string x, the resulting structure is denoted $\hat{T}(x)$ and called the *Minimal Augmented Suffix Tree* of x. It is not difficult to build $\hat{T}_x$ in $O(n^2)$ time and space by embedding the necessary weighting as part of the iterated suffix insertion procedure, hence at an expected cost of $O(n \log n)$ [34]. The time required by the construction given in [36] is instead $O(n \log^2 n)$ in the worst case. The number of auxiliary nodes was bounded by $O(n \log n)$ in [36]. A tighter $O(n)$ bound is implied by recent developments in [6].

III. Implementing the Data Structures

When it comes to the actual allocation in memory of a suffix tree, one faces a number of design choices, prominent among which those pertaining to the implementation of nodes. There are three main possibilities in this regard:

- The node is implemented as an array of size $|\Sigma|$. This yields fast searches, but is likely to introduce an unbearable amount of waste even for small alphabets.
- The node is implemented as a linked list (or, better, as a balanced search tree). This keeps space to a minimum, but introduces an overhead on the search.
- The adjacency of a node is realized as part of a global hash coding. This yields expected constant time search within overall $\Theta(n \log n)$ space.

In our case the space is of high practical concern, so that we use the linked list. Figure 2 displays the minimal augmented suffix tree of our example textstring. As is customary, the substrings representing edge labels are not stored explicitly in the nodes but rather encoded each by an ordered pair of integers to a unique common copy of x, so as to achieve overall linear space. However, even linear space can be problematic: at 20 bytes per node and with a number of nodes 1.5 times the number of symbols in the input string, as typically featured in our experiments, a text of size n needs approximately $30n$ bytes of storage space. In general, although the size of the suffix tree depends on the particular implementation, one might expect it to be never lower than 20 bytes per input symbol (or *bps*) in the worst case. We refer to [37] for a comparative study of various space-efficient allocations.

In general, these space savings are achieved at the expense of higher complexity in either construction, or searching, or both: thus, for instance, the suffix array and the PAT tree need $O(n \log n)$ time for the construction ($O(n)$ on average for the array) and $O(|w| + \log n)$ when searching for a string w.

We use $<w>$ to denote the node, if it exists, precisely at the end of the path in $\hat{T}_x$ labeled by the string w. If instead w ends in the middle of an arc then $<w>$ denotes the node corresponding to the shortest extension of w that ends in a node. In our realization, $<w>$ contains the following items:

- two indices $[i, j]$ identifying an occurrence of w in x, i.e., such that $w = x[i, j]$;
- one pointer to the list of children and one to the list of siblings of $<w>$;
- one counter to store the number of nonoverlapping occurrences of w in x.

The data structure allocating the textstring x should support somewhat contrasting primitives such

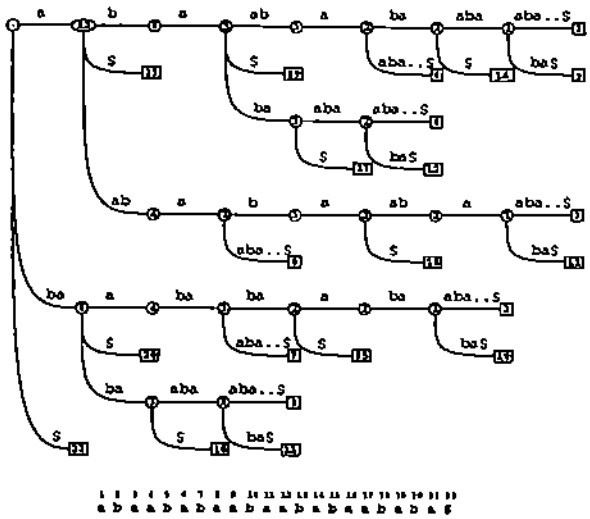

Fig. 2. The minimal augmented suffix tree for abaababaabaababaababa$ shows node additions and weights needed to change a suffix tree into an index for statistics without overlaps. The label of a leaf is the starting position of the suffix corresponding to that leaf. The label of a node is the number of occurrences without overlap of the string on the path ending at that node or anywhere in the middle of the preceding arc

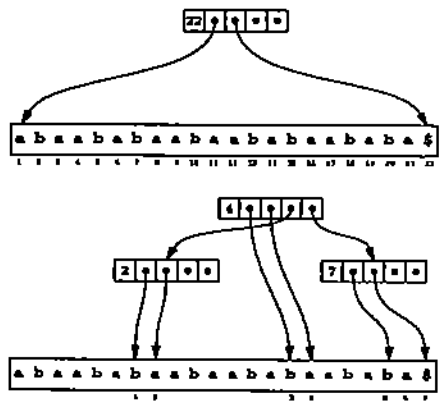

Fig. 3. The data structure allocating textstring abaababaabaababaababa$ prior to and after the removal of aba

as, for instance, efficient string searching and repeated substring deletions. To accommodate the repeated contractions of x, the latter is maintained in a linked list of dynamic arrays, as follows. At the beginning, the text is read from the source into a single array of length n. Subsequently, the removal of the occurrences of a substring w = aba will partition the array into linked fragments, as shown in figure 3. These arrangements are complemented by refresh cycles that will recombine the text in a single array, from time to time, to counteract excessive fragmentations.

Repeatedly building the suffix tree at each stage exacts a considerable toll irrespective of the method adopted. Ideally, one would like to build the tree once and then maintain it, together with updated statistics, following every substring selection and removal. Linear time algorithms for dynamically maintaining the tree under deletion of a string were originally proposed by McCreight together with his construction. Similar problems have been studied by Fiala and Green [19] in the context of sliding window compression. More recently, Larsson [38] showed that the algorithm by Ukkonen can be easily extended to accommodate the sliding window update of the suffix tree in amortized linear time. Gu *et al.* [39] introduced a new data structure for dynamic text indexing that supports insertion and deletion of a single character in $O(\log n)$ time and the i updates involving a substring w that occurs occ_w times in $O(|w| + occ_w \log i + i \log |w|)$. Several recent efforts address the dynamic maintenance of tries of various nature. However, we did not find an existing satisfactory solution to the problem of quickly modifying our statistical index so as to reflect the deletion from the corresponding textstring of *all* the occurrences of a given substring. In our experiments, every new version of the suffix tree was built from scratch. In a later section, we present some heuristics designed to alleviate such computation efforts.

IV. Choosing and Computing a Gain Measure

By "gain measure", we refer here to the function G that drives, at each stage, the selection of the substring that yields the highest compression. In practice, it is not easy to define precisely such a measure, as we explain below.

The main difficulty is due to the fact that at the time when we need to compute the contraction that would be induced by a particular substring, we lack some important costs such as those associated with the optimal encodings of pointers or integers, which can be computed precisely only at the outset. Letting $l(i)$ represent the number of bits needed to encode integer i, we assume for simplicity $l(i) = \lceil \log i \rceil$ at the time the gain is computed. Note that this choice does not affect the appraisal of final compression, the latter being based on purely empirical measures. Along the same lines, one could choose an expression for l that reflects more accurately the efficient encoding of integers in an unknown range [4], [5], [7]. However, as long as the ultimate encoding of the compressed string is not based on those representations, but rather on some statistical treatment (e.g., Huffman encoding), there is hardly any sense in resorting

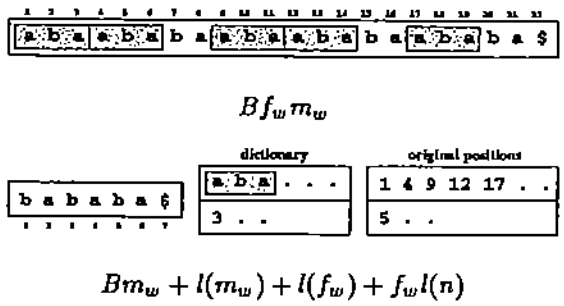

Fig. 4. Illustrating Scheme 1 with $w = \mathtt{aba}$

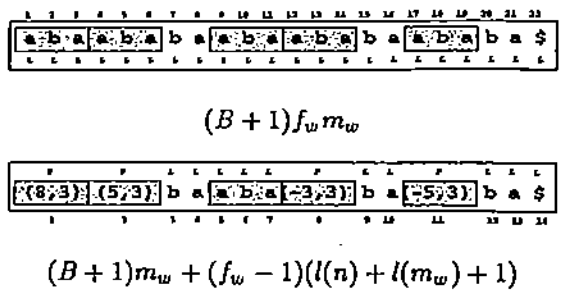

Fig. 5. Illustrating Scheme 2 for $w = \mathtt{aba}$; L and P are used to mark literals and pointers, respectively; the pair (p, l) denotes a pointer to a reference copy that has length l and that is p symbols away

to them and hardly any way to compute $l(i)$ accurately at this stage.

With this choice made, we describe now in succession three possible measures of gain. Let us denote by w the word that maximizes G at some iteration and therefore is selected to be replaced by pointers.

In Scheme 1, we assume that *all* the f_w occurrences of the string w are removed from the text, while w itself is saved in an auxiliary data structure that contains:

- the string w, that is Bm_w bits long, where $m_w = |w|$ and $B = \log|\Sigma|$;
- the length m_w of w, at a cost of $l(m_w)$ bits;
- the value of f_w, at a cost of $l(f_w)$ bits;
- the f_w positions of w in x, at a global cost bounded by $f_w l(n)$ bits.

Figure 4 shows the original and compressed representations for the textstring and the corresponding associated costs. The expression underneath the top figure represents the original cost of the occurrences of w as plain text: with $|w| = m_w$, the f_w copies of w require $Bf_w m_w$ bits in the plain text. In practice, the value of B is appraised based on the empirical entropy of the source: the plain text is encoded with the best statistical encoder from Table I, and then B is set to the average length of a symbol.

The expression at the bottom of the figure is the cost of representing the same occurrences of w based on the external dictionary and auxiliary structures. The difference between the two expressions defines $G_1(w)$ as follows

$$\begin{aligned} G_1(w) &= Bf_w m_w - Bm_w - l(m_w) - l(f_w) - f_w l(n) \\ &= (f_w - 1)Bm_w - l(m_w) - l(f_w) - f_w l(n) \end{aligned}$$

In Scheme 2, we assume that one of the f_w copies of w is kept in the original text, marked by a "literal identification" bit, while the remaining $f_w - 1$ copies are encoded by pointers, each pointer being preceded by a suitable identification bit.

Because of the additional bit, the plain text representation of all the occurrences of w requires $(B+1)f_w m_w$ bits. The pointer-based representation costs are as follows (see Figure 5):

- $(B+1)m_w$ bits for the original copy of w,
- $(f_w - 1)(l(n) + l(m_w) + 1)$ bits for the $f_w - 1$ pointers.

The difference of these expressions defines G_2 as follows

$$\begin{aligned} G_2(w) &= (B+1)f_w m_w - (B+1)m_w - \\ &\quad (f_w - 1)(l(n) + l(m_w) + 1) \\ &= (f_w - 1)(B+1)m_w - \\ &\quad (f_w - 1)(l(n) + l(m_w) + 1) \\ &= (f_w - 1)((B+1)m_w - l(n) - l(m_w) - 1) \end{aligned}$$

In Scheme 3, words in the textfile are replaced by pointers to their corresponding entries in an external dictionary. Thus, following the selection of w at the generic iteration, w is added as a new entry into the dictionary and *all* of its occurrences become pointers to that entry. Also in this case, an auxiliary bit-vector is required in general in order to distinguish between pointers and literals at the outset, both in the text and in the dictionary. However, if pointer recursion is forbidden as we assume in our construction, then the words in the dictionary cannot contain pointers, and a bit-vector is not needed there.

The plain text representation of all the occurrences of w requires $(B+1)f_w m_w$ bits. The costs of the pointer-based representation are (see Figure 6):

- Bm_w bits for the string w in the dictionary,
- $l(m_w)$ to store the length m_w,
- $l(d)f_w$ for the f_w pointers inside the text, where d is the size of the dictionary.

The corresponding expression for $G_3(w)$ is then

$$\begin{aligned} G_3(w) &= (B+1)f_w m_w - Bm_w - \\ &\quad (l(d) + 1)f_w - l(m_w) \\ &= B(f_w - 1)m_w + f_w m_w - \\ &\quad (l(d) + 1)f_w - l(m_w) \end{aligned}$$

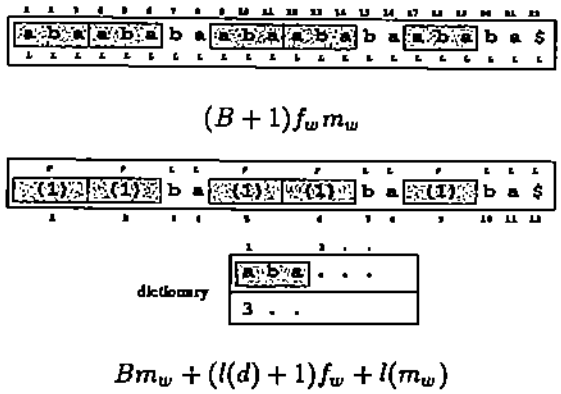

Fig. 6. Illustrating Scheme 3 for $w = \text{aba}$; d is the size of the dictionary, "(1)" denotes a pointer to the first entry in the dictionary, L and P mark literals and pointers respectively

We point out that, for any of the above specifications of G and any word w in x, $G(w)$ is a monotone increasing function of m_w. Moreover, the maximum number of nonoverlapping occurrences of w in x does not change in the middle of an arc of $\hat{T}_x$. Therefore, the word maximizing the gain at each stage *always* ends on a node of $\hat{T}_x$. If now w is this word, then its occurrences are suitably encoded, and the whole process is repeated until the gain becomes zero or negligible, according to some predetermined threshold.

id	*encoding type*
0	Plain
1	Huffman
2	Arithmetic
3	Deflate (ZLIB)
4	RLE
5	RLE + Huffman
6	RLE + Arithmetic
7	RLE + Deflate (ZLIB)

TABLE I
STATISTICAL ENCODERS AVAILABLE TO OFF-LINE

The three schemes just described were embedded in as many encoders, respectively called OFF-LINE$_1$, OFF-LINE$_2$ and OFF-LINE$_3$. As an example, the iterations of OFF-LINE$_1$ are highlighted in Figure 7. The first iteration results in the choice of aba; the second, of ba. The collection of data representing the output encoding appears in the bottom part of the figure. Several implementation details will be given later.

```
1: abaababaabaababaababa$ Substituted substring: "aba"
2: bababa$                Substituted substring: "ba"
------------------------- Final encoding:
sublen = [3 2]
substr = [ababa]
abspol = [0 0]            abspoh = [0 0]
relpol = [0 2 0 2 0 0]    relpoh = [0 0 0 0 0 0]
occurr = [6 3]
text   = [$]
```

Fig. 7. A run of OFF-LINE$_1$ on our example textstring

	paper2 (82,199)		mito (78,521)	
encoder	*size*	*time*$_{[min]}$	*size*	*time*$_{[min]}$
OFF-LINE$_1$	30,848	3.21	16,426	1.66
OFF-LINE$_2$	33,757	3.01	17,741	2.24
OFF-LINE$_3$	30,219	2.38	16,086	2.38

TABLE II
A FIRST GLANCE AT THE THREE OFF-LINE ENCODERS' PERFORMANCES ON A 300MHZ SOLARIS MACHINE

V. RESULTS

The encoders described in the previous section were coded in C++ using the Standard Template Library (STL) [40] and extensively tested. Table II offers a first glance of the performances of the three encoders on two typical inputs, namely, paper2 from the Calgary Corpus and mito, the mitochondrial DNA sequence of the yeast (*Saccharomyces Cerevisiae*). Running times are in the order of 2 or 3 minutes for files of about 80 KB on a 300 Mhz machine running under Solaris. In terms of compression, the best encoder is OFF-LINE$_3$, followed by OFF-LINE$_1$ and, at some distance, OFF-LINE$_2$.

Table III compares performances among textual substitution methods over the entire Calgary Corpus. OFF-LINE$_3$ outperforms the other two encoders on most inputs. As a whole, OFF-LINE encoders perform better than the rest on most inputs, and loose marginally to GZIP where they do. However, a thoroughly faithful comparison to GZIP is made difficult by the many heuristics embedded in that program.

Crossing the boundary of textual substitution methods, the recent block-sorting techniques called BZIP and BZIP2, based on [41], outperform GZIP and OFF-LINE on the whole Calgary Corpus (see Table IV). As seen next, a different scenario is displayed when we turn to biological data sets.

We compare the performance of OFF-LINE encoders with those of standard compression programs in the Table V. The encoder OFF-LINE$_3$ outperforms each and every general purpose encoder on the fourteen chromosomes and the mitochondrial DNA of the yeast. It should be noted that the actual compressions are very small and sometimes negative.

File	*Size* (bytes)	*Huffman* PACK	*LZ-78* COMPRESS	*LZ-77* GZIP	OFF-LINE$_1$	OFF-LINE$_2$	OFF-LINE$_3$
bib	111,261	72,868	46,528	35,063	36,145	39,226	34,442
book1	768,771	438,487	332,056	313,376	305,185	323,007	298,735
book2	610,856	368,423	250,759	206,687	203,249	216,494	204,703
geo	102,400	72,836	77,777	68,493	68,229	69,983	68,726
news	377,109	246,516	182,121	144,840	141,257	150,462	143,246
obj1	21,504	16,330	14,048	10,323	10,845	11,271	11,088
obj2	246,814	194,378	128,659	81,631	88,179	93,915	87,574
paper1	53,161	33,457	25,077	18,577	19,994	21,607	19,289
paper2	82,199	47,731	36,161	29,753	30,848	33,757	30,219
pic	513,216	106,737	62,215	56,442	52,036	55,427	50,885
progc	39,611	26,030	19,143	13,275	14,758	15,527	14,127
progl	71,646	43,093	27,148	16,273	18,508	18,919	16,153
progp	49,379	30,328	19,209	11,246	12,890	13,282	11,160
trans	93,695	65,343	38,240	18,985	21,170	21,170	19,662

TABLE III

THE VARIANTS OF OFF-LINE AGAINST THE OTHER TEXTUAL SUBSTITUTION COMPRESSORS, ON THE CALGARY CORPUS

File	*Size* (bytes)	*BWT* BZIP	*BWT* BZIP2	OFF-LINE$_1$	OFF-LINE$_2$	OFF-LINE$_3$
bib	111,261	27,097	27,467	36,145	39,226	34,442
book1	768,771	230,247	232,598	305,185	323,007	298,735
book2	610,856	155,944	157,443	203,249	216,494	204,703
geo	102,400	57,358	56,921	68,229	69,983	68,726
news	377,109	118,112	118,600	141,257	150,462	143,246
obj1	21,504	10,409	10,787	10,845	11,271	11,088
obj2	246,814	76,017	76,441	88,179	93,915	87,574
paper1	53,161	16,360	16,558	19,994	21,607	19,289
paper2	82,199	24,826	25,041	30,848	33,757	30,219
pic	513,216	49,422	49,759	52,036	55,427	50,885
progc	39,611	12,379	12,544	14,758	15,527	14,127
progl	71,646	15,387	15,579	18,508	18,919	16,153
progp	49,379	10,533	10,710	12,890	13,282	11,160
trans	93,695	17,561	17,899	21,170	21,170	19,662

TABLE IV

COMPARING OFF-LINE WITH CONTEXT-SORTING ENCODERS ON THE CALGARY CORPUS

File	*Size* (bytes)	*Huffman* PACK	*LZ-78* COMPRESS	*LZ-77* GZIP	*BWT* BZIP	*BWT* BZIP2	OFF-LINE$_1$	OFF-LINE$_2$	OFF-LINE$_3$
chrI	230,195	63,144	62,935	66,264	61,674	62,373	57,098	58,631	56,915
chrII	813,137	222,597	219,845	236,837	218,463	221,032	201,617	203,456	201,180
chrIII	315,344	86,281	86,009	91,827	84,809	85,705	77,916	78,983	77,764
chrIV	1,522,191	416,516	409,957	440,056	407,799	411,250	371,230	374,413	370,796
chrV	574,860	157,415	155,944	167,749	154,580	155,731	142,364	143,775	141,919
chrVI	270,148	74,077	73,873	78,925	72,838	73,651	67,451	68,151	67,391
chrVII	1,090,936	298,680	294,417	317,282	293,079	296,245	270,051	272,972	269,265
chrVIII	562,638	154,110	152,265	163,135	151,240	152,992	139,588	140,924	139,271
chrIX	439,885	120,669	118,965	127,805	118,182	119,553	109,507	110,871	109,303
chrX	745,443	204,152	201,783	216,148	200,325	202,223	184,709	186,471	184,287
chrXI	666,448	182,377	180,100	194,119	179,306	180,901	165,780	166,752	165,478
chrXII	1,078,171	295,441	291,754	305,653	288,112	290,800	260,172	261,346	259,898
chrXIII	924,430	253,176	249,099	267,127	248,450	250,735	228,233	231,474	227,610
chrXIV	784,328	215,020	212,219	228,757	210,988	212,816	195,291	196,719	194,947
chrXV	1,091,282	298,762	294,921	317,971	293,838	297,279	270,626	273,366	269,921
chrXVI	948,061	286,579	264,113	278,651	254,947	257,590	234,099	237,365	233,150
mito	78,521	18,149	17,890	19,369	17,962	18,094	16,426	17,741	16,086

TABLE V

COMPARING OFF-LINE WITH OTHER COMPRESSION PROGRAMS ON THE CHROMOSOMES OF THE YEAST

encoder	size	bpc
GZIP	91,827	2.33
PACK	86,281	2.19
COMPRESS	86,009	2.18
BZIP2	85,705	2.17
BZIP	84,809	2.15
OFF-LINE$_3$	77,764	1.97
CDNA [28]	76,471	1.94
BIOCOMPRESS2 [27]	75,682	1.92
AED [42]	75,407	1.913

TABLE VI

COMPARING OFF-LINE WITH DNA-SPECIFIC COMPRESSION PROGRAMS ON THE THIRD CHROMOSOME (CHRIII) OF THE YEAST (315,344 BPS). THE PARAMETER *bpc* REPRESENTS THE AVERAGE NUMBER OF BITS PER CHARACTER IN THE COMPRESSED REPRESENTATION (SOME FINAL SIZES ARE EXTRAPOLATED FROM TABLE 1 OF [42]).

In fact, raw biological sequences (notably, those coming from coding regions [16]) are known to be hard to compress. However, even comparing our encoders with programs specifically designed to compress DNA, the difference in performance is not large, as shown in the Table VI.

It is worthwhile to highlight such DNA-specific analyzers and compressors. As mentioned, information theoretic analyses of biological sequences mingle with the very dawn of bioinformatics studies (see, e.g., [23]), but this area has known recently a considerable revival of interest in view of the massive production of genomic sequences of various kinds. In this context, the detection of redundancy serves not only the purpose of achieving more compact descriptors, but also, and perhaps more importantly, may act as a filter of possibly relevant biological functions. The tenet there is that an incompressible string is more random and thus less likely than a repetitive one to carry some biological function.

Due to mutations, errors in the sequencing process, and other biological events, a substantial part of the redundancy present in DNA manifests itself in form of consecutive (*tandem*) repeats of the same word or *motif*, and palindromes. However, such tandem repeats and palindromes are not exact. Rather, they may occur with substitutions, insertions or deletions of symbols. Moreover, palindromes are actually *complemented*, meaning that in the reverse half of the word the base `A` is mirrored by a `T` (and vice-versa), while `C` is mirrored by a `G` (and vice-versa).

Among the recent dedicated approaches to DNA compression, the one by Grumbach and Tahi [26], [27], called BIOCOMPRESS2, extends LZ-77 to catch very distant repeats and complementary palindromes.

Loewenstern and Yianilos [28] consider the problem of computing good estimates of the entropy of DNA sequences by building a PPM-like predictive model. With respect to the original PPM, they extend the context model by allowing mismatches. Their algorithm estimates the parameters of the model, called CDNA, via a learning process that tries to optimize a complex objective function. The general problem is known to be $\mathcal{NP}$-complete, but they devise more realistic approximation schemes.

Allison, Edgoose and Dix propose the most computationally intensive approach to DNA compression [42]. They search for both approximate repeats and approximate palindromes. Their primary purpose is not to compress the text, but rather to model the statistical properties of the data as accurately as possible and to find patterns and structures within them. They build a model with parameters such as the probability of repeats, of the length of repeats, and of mismatches within repeats. The parameters of the model are estimated by an expectation maximization algorithm that takes time $O(n^2)$ at each iteration. Their results may well be taken to represent the current "state of the art", but as said the algorithm is extremely slow.

Finally, we run OFF-LINE$_3$ on families of related and unrelated genetic sequences. Entries in most genetic databases are flat text files containing one or more sequences that are usually functionally related, with some annotations. The `fasta` format is the most commonly used standard for storing and exchanging genetic files. The generic `fasta` file contains one or more blocks. Each block is composed by one or more annotation lines each starting with the symbol >, followed by the genetic sequence.

Table VII shows the results of running OFF-LINE$_3$ on several families of sequences of the yeast genome. The complete dataset is available at `http://www.cs.purdue.edu/homes/stelo/Off-line/`. The file `Spor_All_2x.fasta` is artificially obtained by concatenating `Spor_All.fasta` with itself, in an attempt to probe into extreme cases of inter-sequence correlation [43]. The last two families (8 and 9) are a segment of *all* the upstream regions of the yeast and thus not strongly related. Table VII shows that not only the absolute performance of OFF-LINE, but also its relative advantage over the other methods improves as the input size increases. Likewise, as soon as the input files contain sequences not as strongly related, the improvements, while still present, decay immediately, as shown for files 8 and 9 in the table. The ability to capture distant relationships is enhanced in the comparison with GZIP and BZIP2 as we move from their default window sizes (900Kb in BZIP2)

Family	*Total size* (bytes)	k	*Huffman* PACK	*LZ-78* COMPRESS	*LZ-77* GZIP	*BWT* BZIP2 -9	OFF-LINE$_3$
(1)	25,008	29	$7{,}996_{(11.0\%)}$	$7{,}875_{(9.6\%)}$	$8{,}008_{(11.1\%)}$	$7{,}300_{(2.5\%)}$	7,119
(2)	31,039	36	$9{,}937_{(12.5\%)}$	$9{,}646_{(9.8\%)}$	$9{,}862_{(11.8\%)}$	$9{,}045_{(3.6\%)}$	8,697
(3)	32,871	38	$10{,}590_{(12.2\%)}$	$10{,}223_{(9.0\%)}$	$10{,}379_{(10.4\%)}$	$9{,}530_{(2.4\%)}$	9,301
(4)	54,325	63	$17{,}295_{(14.6\%)}$	$16{,}395_{(9.9\%)}$	$16{,}961_{(12.9\%)}$	$15{,}490_{(4.6\%)}$	14,778
(5)	112,507	130	$36{,}172_{(17.7\%)}$	$33{,}440_{(11.0\%)}$	$33{,}829_{(12.0\%)}$	$31{,}793_{(6.4\%)}$	29,758
(6)	222,453	258	$70{,}755_{(23.2\%)}$	$63{,}939_{(15.0\%)}$	$68{,}136_{(20.3\%)}$	$61{,}674_{(11.9\%)}$	54,317
(7)	444,906	516	$141{,}431_{(53.4\%)}$	$124{,}637_{(47.1\%)}$	$135{,}816_{(51.5\%)}$	$85{,}142_{(22.6\%)}$	65,891
(8)	399,615	191	$121{,}700_{(12.3\%)}$	$115{,}029_{(7.22\%)}$	$115{,}023_{(7.22\%)}$	$112{,}363_{(5.0\%)}$	106,722
(9)	1,001,002	477	$305{,}054_{(11.9\%)}$	$286{,}971_{(6.4\%)}$	$285{,}064_{(5.8\%)}$	$280{,}334_{(4.1\%)}$	268,012

TABLE VII

COMPARING OFF-LINE$_3$ WITH OTHER COMPRESSION PROGRAMS ON FAMILIES OF SEQUENCES OF THE YEAST. THE FIGURES IN PARENTHESES REPORT PERCENTAGE GAINS ACHIEVED BY OFF-LINE$_3$, k IS THE NUMBER OF UPSTREAM SEQUENCES IN EACH FAMILY, INDIVIDUAL SEQUENCE LENGTH IS 800 BPS EXCEPT IN THE LAST TWO ROWS, WHERE IT IS 2,000. THE ALPHABET CONSISTS OF ABOUT 50 SYMBOLS. THE INPUT STRINGS 1-9 CORRESPOND, IN THIS ORDER TO THE FAMILIES OF SPOR_EARLYII.FASTA, SPOR_EARLYI.FASTA, HELDEN_GCH.FASTA, SPOR_MIDDLE.FASTA, HELDEN_ALL.FASTA, SPOR_ALL.FASTA, SPOR_ALL_2X.FASTA, ALL_UP_400K.FASTA, ALL_UP_1M.FASTA.

to smaller sizes. The results, shown in Table VIII, suggest that the relative advantage of OFF-LINE will increase as it will be applied to larger and larger families.

Family	*LZ-77* GZIP -1	*BWT* BZIP2 -1	OFF-LINE$_3$
(6)	$76{,}629_{(29.1\%)}$	$63{,}332_{(14.2\%)}$	54,317
(7)	$153{,}103_{(57.0\%)}$	$126{,}314_{(47.8\%)}$	65,891

TABLE VIII

CONSTRAINING THE COMPETITORS TO WORK ON SMALL WINDOWS ENHANCES THE GAIN OF OFF-LINE. HERE THE INPUT STRINGS 6 AND 7 CORRESPOND, RESPECTIVELY, TO THE FAMILIES OF SPOR_ALL.FASTA, SPOR_ALL_2X.FASTA (CF. TABLE VII FOR THEIR RESPECTIVE STATISTICS).

VI. FINE TUNING AND OTHER IMPLEMENTATION DETAILS

The most time-consuming activity of the compression phase is the construction of the index trie and its annotation with the values of the gain. We employed three heuristics to overcome the high computational demands of a "full-fledged" version of the compressor.

Table X shows the results achieved by one of these heuristics on the basic algorithm, in which more than just one substring selection and substitution is performed between two consecutive updates of the statistical index. Of course, such an approach saves time on one hand, but it risks blurring the perception of the best candidates for substitution. In our implementation, a heap is maintained with the statistical index, containing at each step the Q best words in terms of G, for some chosen value of the parameter Q. Between any two consecutive index reconstructions, the Q strings in the heap are retrieved and used in succession in a contraction step for the text. It is possible at some point that a string from the heap will no longer be found in the contracted text. In fact, part of the words in the heap turn out to be useless in general. In any case, as soon as all words in the heap have been considered, a new augmented trie is built on the contracted text.

As the Table displays, the number of individual substring substitution passes over the text grows with the maximum allowed size of the heap. On the other hand, we spend less and less time building weighted tries. The overall result is, within a wide interval, a considerable speed up with respect to the eager version of OFF-LINE without substantial penalty in compression performance. When the size of the heap becomes too large (approximately $Q > 100$ in our experiments) only a small subset of the words in the heap is used: most of the computational effort is spent in pattern searching, which results in deterioration of both speed and compression.

Whenever one can assume it as being highly unlikely that very long words occur frequently in a text, then building the statistics for *all* the substrings can be a waste of resources. Pruning the tree speeds up considerably the implementation and saves large amounts of memory. Pruning the tree does not mean that we could completely miss the word involved in a long substitution. If the current best substitution is a word w longer than the threshold l, then the encoder will eventually choose some substring of w of length l because that substring occurs without overlap at

l	paper2 size	paper2 $time_{[min]}$	mito size	mito $time_{[min]}$
10	30,986	2.58	17,044	0.29
50	30,664	2.62	16,491	1.32
100	30,636	2.68	16,470	1.38
∞	30,636	19.39	16,470	10.34

TABLE IX

COMPARING THE PERFORMANCE OF OFF-LINE$_1$ FOR DIFFERENT CHOICES OF THE MAXIMUM ALLOWED LENGTH OF A CANDIDATE FOR SUBSTITUTION. WE FIXED MIN_OCC $= 4, l = 4, Q = 10$.

Q	paper2 size	paper2 $time_{[min]}$	mito size	mito $time_{[min]}$
1	30,773	19.70	16,326	7.06
2	30,780	10.36	16,367	4.06
5	30,785	5.06	16,405	2.24
10	30,787	3.21	16,446	1.66
20	30,826	2.39	16,476	1.36
50	30,904	1.97	16,632	1.28
100	30,923	1.86	16,702	1.37
1,000	30,923	1.98	16,702	1.47

TABLE X

PERFORMANCES OF OFF-LINE$_1$ FOR DIFFERENT CHOICES OF THE SIZE OF THE CANDIDATES HEAP. WE FIXED MIN_OCC $= 2$, MIN_LENGTH $= 2$, $l = 100$.

File	Size	OFF-LINE$_1$	OFF-LINE$_2$	OFF-LINE$_3$
bib	111,261	504	634	465
book1	768,771	2997	2857	2990
book2	610,856	2305	2408	2378
geo	102,400	407	473	503
news	377,109	1789	1634	1619
obj1	21,504	125	111	337
obj2	246,814	1219	1207	1055
paper1	53,161	373	475	342
paper2	82,199	506	717	505
pic	513,216	94	125	222
progc	39,611	255	261	308
progl	71,646	312	267	273
progp	49,379	208	210	252
trans	93,695	340	253	318

TABLE XI

ITERATIONS OF THE MAIN LOOP OF OFF-LINE FOR THE CALGARY CORPUS FILES

File	Size	OFF-LINE$_1$	OFF-LINE$_2$	OFF-LINE$_3$
chrI	230,195	78	603	80
chrII	813,137	112	474	128
chrIII	315,344	61	309	68
chrIV	1,522,191	383	1297	441
chrV	574,860	109	276	118
chrVI	270,148	22	226	30
chrVII	1,090,936	144	1009	162
chrVIII	562,638	91	264	102
chrIX	439,885	54	543	63
chrX	745,443	108	376	123
chrXI	666,448	49	302	58
chrXII	1,078,171	444	1443	499
chrXIII	924,430	187	706	212
chrXIV	784,328	24	441	72
chrXV	1,091,282	128	924	147
chrXVI	948,061	193	755	217

TABLE XII

ITERATIONS OF THE MAIN LOOP OF OFF-LINE FOR THE CHROMOSOMES OF THE YEAST

least as many times as w. Table IX shows that the pruned version of OFF-LINE$_1$ at $l = 100$ performs almost ten time faster and achieves exactly the same compression as the version that builds the complete tree.

The collective speed-up gained from these heuristics combined is significant: our original implementation took several hours to compress those files while afterwards it would complete in few minutes. What is even better, the corresponding loss of efficiency in terms of compression is almost negligible.

As documented in some additional tables, a few hundred iterations of the word selection loop of OFF-LINE suffice on inputs of the order of 100,000 symbols. This suggests that dedicated fine-grained parallel architectures of this kind would implement virtually instantaneous encoders for biosequences and general inputs alike. Tables XII and XI show the modest number of iterations of the main loop performed by OFF-LINE on our inputs, which would be negligible in a parallel context. Therefore, the most expensive tasks, represented by the tree constructions, can be limited considerably in a parallel implementation, turning the method into an on-line, even real-time application.

Since the number of iterations performed determines the size of the vocabulary, whence ultimately of pointers, this generates "quantization" phenomena in the neighborhood of certain values that play critical roles in a computer program. Figure 8 displays the sensitivity of the current implementations to pointer encodings at the crossing of one byte. The two curves plot the sizes of the compressed strings mito and paper2, respectively, at all consecutive stages of the iterated substitutions performed by OFF-LINE$_3$. Following a steady increase until iteration 256, the compression starts decreasing as soon as OFF-LINE$_3$ must employ more than one byte to represent a pointer. In addition to this, the erratic shape of the plot for paper2 suggests, with its several local minima, that it is hard at run time to pin down precisely the best moment when to stop the iterations.

VII. CONCLUDING REMARKS

We have presented a small battery of compressors that perform well on all data but especially well on

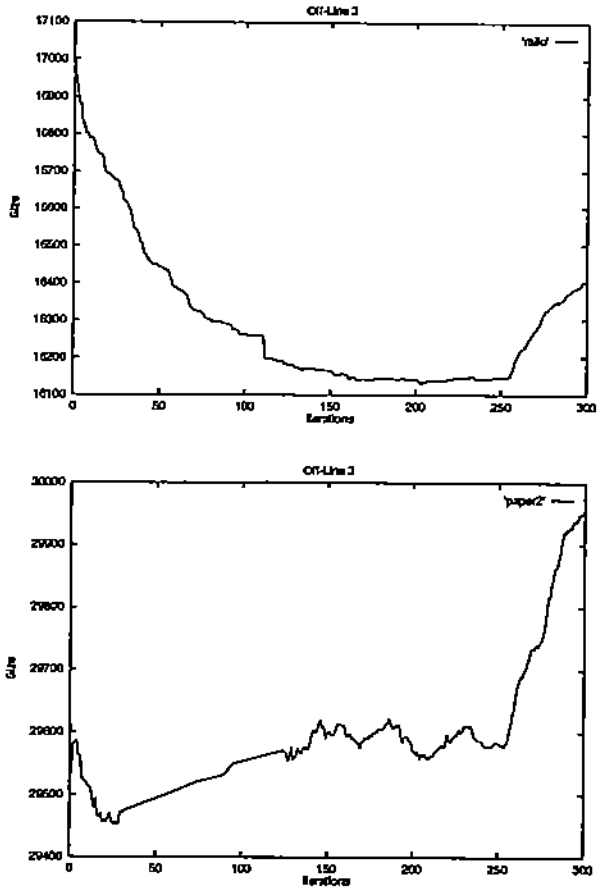

Fig. 8. Compressed sizes of mito (top) and paper2 (bottom) versus number of iterations of OFF-LINE$_3$.

biological data. The basic paradigm is uncluttered, relatively easy to program, and acceptably fast in comparison to ad-hoc, considerably slower and more involved methods.

Besides the obvious challenge of developing versions specifically tailored to biological sequence data, a number of interesting questions emerged in the course of the experiments which would warrant additional effort. These include possible provisions for variable window sizes, better ways to approximate the gain function G, the feasibility and usefulness of reiteration of treatment following the first application of OFF-LINE, and several issues pertaining to the computational efficiency achievable by sequential and parallel implementations. Among the latter, a prominent concern would be to devise efficient algorithms that avoid building the statistical index from scratch after each word selection, and better storage and matching algorithms for our data structure. In fact, as documented in our tables, a few hundred iterations of the word selection loop of OFF-LINE suffice on inputs of the order of 100,000 symbols. This suggests that dedicated fine-grained parallel architectures of this kind would implement virtually instantaneous encoders for biosequences and general inputs alike.

In view of the discussion in the previous Section, it is interesting for a moment to regard OFF-LINE also as a paradigm for inferring hierarchical grammatical structures in sequences. Figure 9 displays the grammar inferred for our example string by the SEQUITUR algorithm by Nevill-Manning *et al.* [17], which is essentially patterned after an LZ parsing scheme. Except for the one involving the start symbol S, productions are constrained to have right-hand sides consisting of digrams. A grammar subtended by the strings of Figure 7 is shown in Figure 10. Re-iteration of the treatment would expose productions of the form $C \rightarrow AAB$ and $D \rightarrow AB$, and finally $S \rightarrow CCD$.

The rationale to build grammar based on some measure of compression can be justified by the "Occam's razor" principle. Occam's razor is the *principle of parsimony* in experimental sciences. In machine learning, its expressed goal is to discover the simplest hypothesis (or model) that is consistent with the training data. In this context, the grammar that our encoder is looking for is the shortest "explanation" of the original string in terms of information content.

S → DDC$
A → ba
B → aA
C → BA
D → BC

Fig. 9. Hierarchical grammar produced by SEQUITUR for abaababaabaababaababa$

S → AABAABAB $
A → aba
B → ba

Fig. 10. First layer of grammar produced by OFF-LINE

Acknowledgements. We are thankful to E. Rivals, J. Storer and F. Tahi for helpful discussions.

REFERENCES

[1] T. C. Bell, J. G. Cleary, and I. H. Witten, *Text Compression.*, Prentice Hall, 1990.
[2] J. A. Storer, *Data Compression: Methods and Theory*, Computer Science Press, 1988.
[3] J. A. Storer and T. G. Szymanski, "Data compression via textual substitution," *Journal of the ACM*, vol. 29, no. 4, pp. 928–951, Oct. 1982.
[4] A. Apostolico and A. Fraenkel, "Robust transmission of unbounded strings using Fibonacci representations,"

IEEE Transactions on Information Theory, vol. 33, no. 2, pp. 238–245, 1987.

[5] S. Even and M. Rodeh, "Economical encoding of commas between strings," *Communication of the ACM*, vol. 21, no. 4, pp. 315–317, Apr. 1978.

[6] A. S. Fraenkel and J. Simpson, How many square can a string contain, *J. Comb. Theory Ser. A* 82, 112-120, 1998.

[7] P. Elias, "Universal codeword sets and representations of the integers," *IEEE Transactions on Information Theory*, vol. 21, pp. 194–202, 1975.

[8] A. Lempel and J. Ziv, "On the complexity of finite sequences," *IEEE Trans. on Inform. Theory*, vol. 22, pp. 75–81, 1976.

[9] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Trans. on Inform. Theory*, vol. IT-23, no. 3, pp. 337, May 1977.

[10] J. Ziv and A. Lempel, "Compression of individual sequences via variable-rate coding," *IEEE Trans. on Inform. Theory*, vol. 24, no. 5, Sept. 1978.

[11] M. Crochemore and W. Rytter, "Efficient parallel algorithms to test square-freeness and factorize strings," *Inform. Process. Lett.*, vol. 38, pp. 57–60, 1991.

[12] L. M. Stauffer and D. S. Hirschberg, "PRAM algorithms for static dictionary compression," in *Proceedings of the 8th International Symposium on Parallel Processing*, Howard Jay Siegel, Ed., Los Alamitos, CA, USA, Apr. 1994, pp. 344–348, IEEE Computer Society Press.

[13] S. De Agostino and J. A. Storer, "On-line versus off-line computation in dynamic text compression," *Inform. Process. Lett.*, vol. 59, no. 3, pp. 169–174, 1996.

[14] K. S. Fu and T. L. Booth, "Grammatical inference: Introduction and survey – Part I," *IEEE Trans. on Systems, Man and Cybernetics*, vol. 5, pp. 95–111, 1975.

[15] K. S. Fu and T. L. Booth, "Grammatical inference: Introduction and survey – Part II," *IEEE Trans. on Systems, Man and Cybernetics*, vol. 5, pp. 112–127, 1975.

[16] C. Nevill-Manning, and I. H. Witten. Protein is incompressible. In J. A. Storer and M. Cohn, eds., *Data Compression Conference*, pages 257–266, Snowbird, Utah, 1999.

[17] C. Nevill-Manning, Ian H. Witten, and D. Maulsby, "Compression by induction of hierarchical grammars," in *DCC: Data Compression Conference*. 1994, pp. 244–253, IEEE Computer Society TCC.

[18] A. Apostolico and E. Guerrieri, "Linear time universal compression techniques based on pattern matching," in *Proceedings of the 21-st Allerton Conference on Communication, Control and Computing, Monticello, Ill.* 1983, pp. 70–79, University of Illinois Press.

[19] E. R. Fiala and D. H. Greene, "Data compression with finite windows," *Communications of the ACM*, vol. 32, pp. 490–505, 1989.

[20] R. N. Horspool, "The effect of non-greedy parsing in Ziv-Lempel compression methods," in *DCC: Data Compression Conference*. 1995, pp. 302–311, IEEE Computer Society TCC.

[21] Frank Rubin, "Experiments in text file compression," *Communications of the ACM*, vol. 19, no. 11, pp. 617–623, Nov. 1976.

[22] N. Jesper Larsson and Alistair Moffat, "Offline dictionary-based compression," in *Proceedings of the IEEE Data Compression Conference*, Mar. 1999, pp. 296–305.

[23] L. Gatlin, *Information Theory and the Living Systems.*, Columbia University Press, 1972.

[24] L. Allison and C.N. Yee, "Minimum message length encoding and the comparison of macro-molecules.," *Bull. Math. Biol. '52*, pp. 431–453, 1990.

[25] M. Farach, M. Noordewier, S. Savari, L. Shepp, A. Wyner, and J. Ziv, "On the entropy of DNA: Algorithms and measurements based on memory and rapid convergence," in *ACM-SIAM Annual Symposium on Discrete Algorithms*, San Francisco, California, 22–24 Jan. 1995, pp. 48–57.

[26] S. Grumbach and F. Tahi, "Compression of DNA sequences," in *Data Compression Conference*, J. A. Storer and M. Cohn, Eds., Snowbird, Utah, 1993, pp. 340–350.

[27] S. Grumbach and F. Tahi, "A new challenge for compression algorithms: genetic sequences," *Inform. Proc. and Mngm.*, vol. 30, no. 6, pp. 875–886, 1994.

[28] D. M. Loewenstern and P. N. Yianilos, "Significant lower. entropy estimates for natural DNA sequences," in *Data Compression Conference*, J. A. Storer and M. Cohn, Eds., Snowbird, Utah, 1997, pp. 151–160.

[29] D. M. Loewenstern, H. M. Berman, and H. Hirsch, "Maximum a posteriori classification of DNA structure from sequence information," *Pacific Symp. Biotech.*, Jan. 1998.

[30] D. M. Loewenstern, H. Hirsh, P. Yianilos, and M. Noordewier, "DNA sequence classification using compression-based induction," Tech. Rep. 95-04, DIMACS, Apr. 1995.

[31] A. Milosavljevic and J. Jurka, "Discovery by minimal length encoding: a case study in molecular evolution.," *Machine Learning*, vol. 12, pp. 69–87, 1993.

[32] E. Rivals, J. P. Delahaye, M. Dauchet, and O. Delgrange, "A guaranteed compression scheme for repetitive DNA sequences," in *Data Compression Conference*, J. A. Storer and M. Cohn, Eds., Snowbird, Utah, 1996, p. 453.

[33] E. Rivals, O. Delgrange, J. P. Delahaye, M. Dauchet, M. O. Delorme, A. Henaut, and E. Ollivier, "Detection of significant patterns by compression algorithms: the case of approximate tandem repeats in DNA sequences," *CABIOS*, vol. 13, no. 2, pp. 131–136, 1997.

[34] A. Apostolico and W. Szpankowski, "Self-alignment in words and their applications," *J. Algorithms*, vol. 13, no. 3, pp. 446–467, 1992.

[35] E. Ukkonen, "On-line construction of suffix trees," *Algorithmica*, vol. 14, no. 3, pp. 249–260, 1995.

[36] A. Apostolico and F. P. Preparata, "Data structures and algorithms for the strings stastitics problem," *Algorithmica*, vol. 15, no. 5, pp. 481–494, May 1996.

[37] S. Kurtz, "Reducing the space requirments of suffix trees," Tech. Rep. 98-03, Technischen Fakultät, Universität Bielefeld, 1998.

[38] N. J. Larsson, "Extended application of suffix trees to data compression," in *DCC: Data Compression Conference*. 1996, pp. 190–199, IEEE Computer Society TCC.

[39] M. Gu, M. Farach, and R. Beigel, "An efficient algorithm for dynamic text indexing," in *Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, Arlington, VA, 1994, pp. 697–704.

[40] D. R. Musser and A. A. Stepanov, "Algorithm-oriented generic libraries," *Software–Practice and Experience*, vol. 24, no. 7, pp. 623–642, July 1984.

[41] M. Burrows and D. J. Wheeler, "A block-sorting lossless data compression algorithm," Tech. Rep. 124, Digital Equipments Corporation, May 1994.

[42] L. Allison, T. Edgoose, and T. I. Dix, "Compression of strings with approximate repeats.," *Intell. Sys. in Mol. Biol. '98*, pp. 8–16, June 1998.

[43] Jon Bentley and Douglas McIlroy, "Data compression using long common strings," in *Proceedings of the IEEE Data Compression Conference*, Mar. 1999, pp. 287–295.

[44] S. Lonardi, *Off-Line Data Compression by Textual Substitution*, Ph.D. thesis, Università di Padova, Dipartimento di Ingegneria Elettronica e Informatica, February 1999.