Compression of DNA sequences *

Extended Abstract

Stéphane Grumbach Fariza Tahi
I.N.R.I.A.
Rocquencourt, BP 105
78153 Le Chesnay France

February 10, 1994

Abstract

We propose a lossless algorithm to compress the information contained in DNA sequences. None of the available universal algorithms compress such data. This is due to the specificity of genetic information. Our method is based on regularities, such as the presence of palindromes, in the DNA. The results we obtain, although not satisfactory, are far beyond classical algorithms.

1 Introduction

There are plenty of specific types of data which need to be compressed, for ease of storage and communication. Among them are texts (natural language, programs, etc.), images, sounds, etc. In this paper, we focus on the compression of a specific kind of texts only, namely DNA sequences. The deoxyribonucleic acid (DNA) constitute 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 (EMBL, GenBank, DDJB) are developed around the world to store nucleotide sequences (DNA, RNA) and amino-acid sequences of proteins. It is well known that their size increases nowadays exponentially fast. Not as big yet as some other scientific databases, their size is in hundreds of GB [Kam91]. The compression of genetic information therefore constitutes a very important task.

DNA sequences can be considered as texts over a four letter alphabet, $\{A, C, G, T\}$. For complete genomes, these texts can be very long. The human genome for instance contains three billions characters over twenty-three pairs of chromosomes. It contains all the genetic material of the human beings. Only about ten percent of its sequence contains genes. The rest is considered to be non coding.

*Work supported in part by the French agency for research on human genome, GREG.
E-mail addresses: stephane.grumbach@inria.fr, fariza.tahi@inria.fr.

Numerous algorithms have been proposed to compress the size of texts [LH87, Sto88]. There are two main approaches, the statistical and the substitutional. In the first one, blocks of fixed length (generally letters) are encoded with respect to their probability of appearance; short (long) codewords are used for (non) frequent patterns [Huf52]. In the second one, factors of different length are encoded using a pointer to their previous occurrences in the text [LZ76].

Unfortunately, the compression of DNA sequences appears to be a difficult task. They are at first glance very similar to random strings, and have only very hidden regularities. The classical algorithms for text compression do not work on DNA sequences. Worse, they extend the content of the sequences. Indeed, an ASCII file containing a DNA sequence is generally compressed at a rate of less than 74% by both *compact* and *compress* provided on Unix, while the sequence can be stored in a file of 25% of its initial size by a simple encoding of the four characters on two bits versus one byte.

We define an algorithm *biocompress* in the spirit of Ziv and Lempel based on the detection of factors and palindromes. It seems less relevant to define a probability based algorithm to compress DNA sequences. This is due to the similar probability of appearance of blocks of the same length. *biocompress* is a lossless compression algorithm. The compression rate is favored in the trade-off compression-rate/resources-used. This makes sense for genetic sequences which should be compressed once when they are accepted in a database. On the other hand, the decompression is efficient. Its success is due to the fact that it can only compress (i.e. the encoding never increases the input size), it detects factors arbitrarily long and far from each other, and it detects palindromes.The compression rates obtained with *biocompress* are not completely satisfactory yet, but it improves drastically the rates of universal algorithms.

We propose two compression modes, horizontal and vertical. In the horizontal mode, a DNA sequence is compressed into a shorter form. For some genes, the rate is higher than 30%. In the vertical mode, a DNA sequence A is compressed with respect to another sequence B. The output of *biocompress* contains the information required to construct the sequence A from the sequence B. This makes sense if A and B are very similar. Genetic databases contain large amounts of close sequences such as the same gene in different species (the gene of the globin is a well known example for instance). A sequence of reference may be stored in the database while other sequences are stored in a compressed form with respect to it.

Compression of DNA sequences is also interesting from a theoretical point of view. Indeed, in the horizontal mode, it leads to a concept of entropy of genetic sequences. Coding and non-coding regions in genes for instance have already been separated by statistical analysis of the periodicity of the alternance of purines $\{A, G\}$ and pyrimidines $\{C, T\}$ in the sequences [AM90]. The entropy resulting from the degree of compression may provide new insights in the separation of coding and non-coding regions. In the vertical mode, it may lead to an organization of the data related to the similarities between the species.

The paper is organized as follows. In the next section, we review some knowledge on genetic sequences and present the regularities which are used to compress them. In

Figure 1: Secondary Structure

section 3, we introduce a new algorithm for DNA compression, *biocompress*. Finally, we analyze our results on different genetic inputs in the last section.

2 Genetic Data

In this section, we recall briefly some characteristics of DNA sequences which are relevant for their compression. Due to space limitation, we only sketch the purely biological aspects. Nevertheless, it is important to understand where the regularities arise. There are obviously many other regularities, but the following ones are used in the algorithm *biocompress*.

The deoxyribonucleic acid (DNA) is a molecule composed of deoxyribonucleotides connected by phosphodiester linkages. There are four kinds of nucleotides: adenine (A), cytosine (C), guanine (G), and thymine (T). There is another kind of nucleic acid in the cells, namely the ribonucleic acid (RNA), containing the same nucleotides but where the thymine is replaced by the uracil (U). The molecules of RNA are folded up in a complicated way. It is their tertiary (i.e. three dimensional) structure which determines their activity. This structure is held together by links between complementary bases, i.e. between G and C, A and U, and possibly between G and U. There are pairs of complementary subsequences of the RNA which are mapped together. Such pairs of subsequences are called *palindromes*, or inverted repeat. The *secondary structure* of an RNA is a planar graph with the complementary base pairs linked together (see Figure 1). The structure of the RNA's is the origin of abundant palindromes in the DNA. Other kinds of palindromes exist also in the DNA such as for instance in pre-gene areas. Palindromes constitute a fundamental primitive in both DNA and RNA sequences. They can be very long, as in the genome of a chloroplast [SOT+86], where there is an inverted repeat of length above 25000 characters. Note also that DNA sequences are always read in the same direction, such as texts in natural or programming languages for instance. Properties of DNA sequences and mathematical tools to study them are presented in [?]. Algorithms to detect palindromes have been studied in [ABG92]. In this paper, we focus on DNA sequences only. The algorithm *biocompress* can also be used on RNA sequences but not on amino-acid sequences of proteins.

The sequences on which we realized compression tests are the following (see Figure 3). The complete genomes of two mitochondries: MIPACGA, the mitochondry of podospora anserina and MPOMTCG, the mitochondry of marchantia polymorpha; the complete genomes of two chloroplasts: CHNTXX, the tobacco chloroplast and MPOCPCG, the marchantia polymorpha (liverwort) chloroplast; the complete sequence of the chromosome III of the yeast, YSCCHRIII, saccharomyces cerevisiae (this is the biggest complete sequence obtained by the biologists); the complete genomes of viruses: VACCG, Vaccinia virus and HS5HCMVCG, the human cytomegalovirus strain AD169 and finally, some human genes, and in particular: HUMGHCSA, the human growth hormone and chorionic somatomammotropin genes and HUMHBB, the human beta globin region on chromosome 11. Similar results as those presented in Figure 3 were obtained on a bunch of other inputs that are not listed here.

We present our algorithm in the next section.

3 The algorithm biocompress

The algorithm is based on the substitution of factors with shorter references to earlier occurrences of identical factors or complementary factors (palindromes). It encodes a text on the four letter alphabet {A, C, G, T} into a binary sequence. It follows the pioneering work of Lempel and Ziv. This two authors proposed two methods. In the first one [ZL77], the occurrences of the factor to encode are searched in a window. The second one [ZL78] is based on a dictionary containing the already encoded factors.

To compress DNA sequences, the first approach is more appropriate. Indeed, methods based on dictionary are generally not complete [Wel84]. They may loose a high percentage of factors. In general, windows have a fixed width. If s is the window's width and i is the current position, the window ranges between the $(i-s)^{ith}$ and the i^{ith} character [SS82, FG89]. The use of small windows is efficient on texts whose redundancy is local. However, in the case of DNA sequences, redundancies may occur at very long distances and factors can be very long. We therefore allow a window of the size of the input.

As shown in the previous section, there are specific redundancies in DNA sequences, namely the palindromes. Such repetitions do not exist in texts in natural languages or programming languages. Let us next define formally the palindromes. Consider a sequence of characters f on the alphabet $\{A, C, G, T\}$:

$$a_1, a_2, ..., a_n$$

f^{-1} denotes the sequence in the inverse order:

$$a_n, a_{n-1}, ..., a_1$$

f_α denotes the sequence obtained from f by mapping each character to its complementary character, A is mapped to T and reciprocally, and C is mapped to G and reciprocally. So, the sequence:

$$A\ A\ C\ G\ T\ T$$

is mapped to the sequence:

$$T\ T\ G\ C\ A\ A.$$

The pair of factors f and f_α^{-1} is called a palindrome. (Remark that the sequence in the above example is a specific kind of palindrome, namely a connected palindrome, with no gaps between the two factors, AAC and GTT.)

The data structure used to recognize the factors is a very simple automaton. More complex automaton have been defined in the literature [Zip92, FG89]. They can hardly be used in this context. Our automaton is a 4-ary complete tree of height h ($h = 8$ in our tests). It recognizes all factors of length less or equal to h on the alphabet $\{A, C, G, T\}$. The positions of these factors are stored in the automaton. Consider a factor leading from the root to a leaf l. If the size of the factor is strictly smaller than h, then only the position of its first occurrence is stored on leaf l. Otherwise, leaf l contains the list of positions of all its occurrences. So, factors of size longer than h can be detected by (i) a search on the tree followed by (ii) a search on the input string. At each step of the compression, the longuest factor beginning at the current position and which occurs in the part of the chain which has been already coded is chosen (See Figure 2). This factor may be either an identical factor or a palindrome. It is then encoded by a pair of integers (l, p), where l is the length of the factor and p the position of its first occurrence. In the following, we assume that we have a binary encoding for the integers. It will be introduced later.

Each character A, C, G, or T can be coded on two bits (by respectively 00, 01, 10, 11). It may happen that the size of the codeword (l, p) associated to a factor is bigger than the size of the factor (where the characters are encoded on two bits) itself. In order to avoid the extension of the sequences, we encode a factor f by the pair (l, p) only if the size of the codeword is not too big relatively to the size of the factor, i.e.

$$\mid (l, p) \mid + c \leq 2l,$$

where $\mid (l, p) \mid$ is the size of the representation of (l, p) and c is a constant proportional to the size of control bits. If no factor satisfies the previous requirement, the first character is transmitted, and the operation is repeated with the factors beginning at the following character. The output code contains therefore two types of codewords: characters on two bits (*literal* codewords) and pairs of integers of the form (l, p) (*copy* codewords). We add an integer in front of each sequence of either characters or pairs of integers only. This will be important to be able to distinguish between the two. It indicates the change of semantics of the encoding.

For example, the following sequence:

$$AACTGTTGTTGTTAGAACTGTTGTT \qquad (*)$$

is translated into the expression:

$$6\ AACTGT\ 1\ (7,4)\ 2\ AG\ 1\ (10,1)$$

The efficiency of the compression depends on the space required for the encoding of the integers. In front of sequences of characters (*literal* codewords), we reserve a variable space. The reason is that we have noticed that the number of characters by literal codewords varies between one and some thousands of characters for big genomes. To encode such integers, we use a prefix code, the Fibonacci code. This code is based on the Fibonacci numbers, which are used as a base. It has been shown that there is a way to represent the integers into sequences of 0 and 1 with no consecutive 1. A representation $n = n_1 n_2 ... n_p$ (note than $n_1 = 1$) is then reversed and a 1 is concatenated at the end. The integer is represented by:

$$n_p n_{p-1} ... n_2 11$$

A Fibonacci codeword is recognized by its end with two consecutive 1's.

Such a Fibonacci number is added in front of every sequence of characters or pairs of integers specifying a group of consecutive copies.

We next present the way the length and the position of a copy codeword are encoded. The encoding is more subtile. The factors in the DNA sequences have often a size smaller than 38 characters. Moreover under seven characters, it is not worth encoding them. We therefore use five bits for the binary encoding of the length l, were 7 is coded by '00000', 8 by '00001', ..., and 38 by '11111'. In the case of factors of length bigger than 38, we encode the first 38 characters on 5 bits ('11111') and the rest of the characters by the Fibonacci encoding. For example, if $l = 45$, we encode it by:

1111101011

where the five last bits encode the integer 7 (7=45-38) in the Fibonacci code.

In classical algorithms based on a window, a fix number of bits is allocated to encode the position of the factor, since the size of the window is fixed. Here, the position is variable (between 1 and $i-1$, were i is the current position). The number of bits in which the position is encoded is variable. We choose dynamically between the Fibonacci codeword and a binary codeword depending on the current position. In the second case, the number of bits allocated is equal to $\lceil log_2(i-1) \rceil$ or $\lceil log_2(i-1) \rceil + 1$, where i is the current position. For example, if the position p is equal to 45, and i equal to 100, the Fibonacci codeword is '001010011', while the binary codeword is '0101101'. Since the binary codeword is shorter than the Fibonacci one, p is encoded by '0101101' on 7=$\lceil log_2(100-1) \rceil$ bits. To distinguish between a Fibonacci codeword and a binary one, we add one bit after the two first consecutive bits '1' if there are any. In the Fibonacci case, we add the bit '0' and in the binary case, we add the bit '1'. So, the position 45 will be encoded on 8 bits (8=7+1) by:

01011101

begin
$i = 1$;
while $(i \leq n)$

do begin

find the longest factor f

starting at a position smaller than i
matching the factor at the current position i

let l_1 be the length and p_1 be the position of f
find the longest palindrome f'

starting at a position smaller than i
matching the factor at the current position i

let l_2 be the length and p_2 be the position of f'
if $\mid (l_1, p_1) \mid + c > 2l_1$ **then**

if $\mid (l_2, p_2) \mid + c > 2l_2$ **then**
output the character at position i encoded on two bits;
$i = i + 1$;

else
output the codeword (l_2, p_2);
$i = i + l_2$;

else

if $(\mid l_1 \mid > \mid l_2 \mid$ **or** $\mid (l_2, p_2) \mid + c > 2l_2)$ **then**
output the codeword (l_1, p_1);
$i = i + l_1$;

else
output the codeword (l_2, p_2);
$i = i + l_2$;

end

end

Figure 2: The algorithm *Biocompress*

During the decompression, if no consecutive 1's have been found in the $\lceil log_2(i-1) \rceil$ bits, then it is binary. Otherwise, it depends on the bit following the two consecutive 1's. Note that the Fibonacci encoding is used only in the case it is smaller than the binary encoding.

The sequence (*) of the example is then encoded in binary notation as follows:

10011 00 00 01 11 10 11 11 00000 100 011 00 10 11 00011 110

(The spaces are left for readability only.)

For the encoding of equal factors, the position which is considered is the absolute position, i.e. the position from the beginning of the chain. If the factor to encode is a palindrome, we consider its relative position, i.e. its position relatively to the current position, because palindromes are often close to each other.

The encoding of the palindromes is done in a similar way. The only difference is that for the dynamic encoding of the position, we allocate $\lceil log_2(i-l) \rceil$ bits (l being the length of the palindrome).

A bit is also used to distinguish simple factors from palindromes. This was omitted in the code of the example. We also consider very close palindromes, when the gap is smaller than 16. There are plenty of them in DNA sequences. A special encoding is defined for them with fixed length for the gap and the factors.

4 Analysis of the Results and Conclusions

We next present the compression rates obtained with *biocompress* on the sequences presented in Section 2. We present open problems and draw some conclusions at the end of this section.

Figure 3 summarizes the compression rates of different algorithms on DNA sequences. *compact* and *compress* are the compression algorithms provided on Unix. *compact* is based on the dynamic Huffman code, while *compress* is based on Ziv-Lempel's method. The *compression rate* is defined as follows. Let I be an input sequence over a k letter alphabet, and O be the output sequence over a k' letter alphabet. Then, the compression rate is:

$$1 - \frac{\mid O \mid}{log_{k'}(k) \mid I \mid}$$

For DNA sequences, $k = 4$. In the case of compact and compress, $k' = 256$, and $log_{k'}(k) = \frac{1}{4}$, while, for *biocompress*, $k' = 2$, and $log_{k'}(k) = 2$.

The compression rate for both compact and compress is always negative. In other words neither of them compresses the information contained in the sequences. They don't even reduce the file size as much as a standard binary encoding of the four letters would have done.

It is important to notice that there is a strong correlation between the compression rates of compact and compress. But there is no obvious correlation between the compression rates of those two algorithms with the rates of *biocompress*. Surprisingly,

		Algorithm		
Sequence	size	*compact*	*compress*	*biocompress*
CHNTXX	155844	-9.32%	-9.36%	16.26%
HS5HCMVCG	229354	-9.95%	-10.65%	5.90%
HUMGHCSA	66495	-12.32%	-9.68%	32.75%
HUMHBB	73323	-10.04%	-9.73%	3.22%
HUMHDABCD	58864	-10.35%	-11.48%	3.86%
MIPACGA	100314	-3.94%	-5.81%	0.48%
MPOCPCG	121024	-4.54%	-3.73%	8.44%
VACCG	191737	-6.79%	-8.37%	7.91%
YSCCHRIII	315357	-9.11%	-9.41%	1.62%

Figure 3: Compression rate

our best result (HUMGHCSA) corresponds to the worst rate of compact, while our worst result (MIPACGA) corresponds to the best rate of compact.

The results of *biocompress* are variable. In general, the palindromes play an important role. This is in particular the case for CHNTXX which contains a very long palindrome. Redundancies occur also far away from each other, and there are only very few local redundancies.

For the vertical compression, i.e. compression of a sequence with respect to another sequence, we have only very partial results yet. In some cases, we get good compression rates (60% for variations of the pattern ALU). The results seem less important for genes, probably because of the numerous gaps in the repetitions.

The compression rate of *biocompress* is not satisfactory yet. We are currently working on extensions in various directions. Repetitions in genes constitute generally *non connected sequences*, i.e. sequences with gaps of variable size. This is the case for instance of sequences such as the ALU sequence in primate genomes, or of genes of histone or genes of rRNA's. This kind of redundancies is not well detected by *biocompress*. The gaps dramatically decreases the efficiency of the algorithm. Another regularity that is of interest is the *secondary structure*. It may be used to encode in the same way sequences having the same secondary structure and some fixed parts. This is not detected by *biocompress* since the parts of the sequences (coding for tRNA's for instance) which are fixed, are of a short length. Compression algorithms are generally used to ease the storage and the communication of data. In the case of DNA sequences, we are thinking of developing alignment algorithms working directly on compressed DNA sequences. This may therefore lead to optimization techniques for alignment algorithms. As a side effect, compression of DNA sequences lead to the detection of regularities. It is interesting to note that there are repetitions and palindromes in DNA sequences but very few pairs of factors f, f^{-1}. This makes sense since a DNA sequence is oriented, but it was confirmed by the experience. It justifies also that we look only for simple factors and palindromes, instead of searching f, f_α, f^{-1} and f_α^{-1}.

Finally, the compression technique introduces formal criteria to analyze DNA sequences which can be related to the computational theory of information, where the complexity of a sequence is defined as the size of the smallest program to generate it [Kol65] (horizontal compression) or as the minimum number of computation steps of the program [Ben88] (vertical compression).

Ackwnoledgments: The authors wish to thank Max Dauchet, Jean-Paul Delahaye, Mireille Régnier and Marc Zipstein for fruitful discussions, and Jean-Jacques Codani and the Généthon for providing us the DNA sequences.

References

[ABG92] A. Apostolico, D. Breslauer, and Z. Galil. Optimal parallel algorithms for periods, palindromes and squares. In *unpublished*, 1992.

[AM90] D. G. Arquès and C. J. Michel. Periodicities in coding and noncoding regions in genes. *Journal of Theoretical Biology*, 143:307–318, 1990.

[Ben88] C. H. Bennett. *The Universal Turing Machine : A Half-Century Survey*, chapter Logical Depth and Physical Complexity, pages 227–257. R. Herken, oxford university press edition, 1988.

[FG89] E. R. Fiala and D. H. Greene. Data compression with finite windows. *Communications of the ACM*, 32(4):490–505, 1989.

[Huf52] D. A. Huffman. A method for the construction of minimum-redundancy codes. In *Proc. IRE*, volume 40, pages 1098–1101, Sep 1952.

[Kam91] N. Kamel. Panel: Data and knowledge bases for genome mapping: What lies ahead? In *Proc. Int. Conf. on Very Large Databases*, 1991.

[Kol65] A. N. Kolmogorov. Three approaches for defining the concept of information quantity. *Information Transmission*, 1:3–11, 1965.

[LH87] D. A. Lelewer and D. S. Hirschberg. Data compression. *ACM Computing Surveys*, 19(3):261–287, 1987.

[LZ76] A. Lempel and J. Ziv. On the complexity of finite sequences. *IEEE Trans. Inform. Theory*, 22(1):75–81, 1976.

[SOT+86] K. Shinozaki, M. Ohme, M. Tanaka, T. Wakasugi, and N. Hayashida. The complete nucleotide sequence of the tobacco chloroplast genome : its gene organisation and expression. *The EMBO Journal*, 5(9):2043–2049, 1986.

[SS82] J. A. Storer and T. G. Szymansk. Data compression via textual substitution. *Journal of the ACM*, 29(4):928–951, 1982.

[Sto88] J. A. Storer. *Data Compression, methods and theory*. Computer science Press, 1988.

[Wel84] T. A. Welch. A technique for high performance data compression. *Computer*, pages 8–19, 1984.

[Zip92] M. Zipstein. Data compression with factor automata. *Theoretical Computer Science*, pages 213–221, 1992.

[ZL77] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, May 1977.

[ZL78] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, Sept 1978.