IT-95-285 **Nov. 95**

Laboratoire d'Informatique Fondamentale de Lille

Publication IT-95-285

A Guaranteed Compression Scheme for Repetitive DNA Sequences

Éric Rivals, Jean-Paul Delahaye, Max Dauchet, Olivier Delgrange

21 Novembre 1995

UNIVERSITE DES SCIENCES ET TECHNOLOGIES DE LILLE
U.F.R. d'I.E.E.A. Bât. M3 – 59655 VILLENEUVE D'ASCQ CEDEX
Tél. 20 43 47 24 Télécopie 20 43 65 66

A Guaranteed Compression Scheme for Repetitive DNA Sequences *

É. Rivals, J-P. Delahaye, M. Dauchet
Lab. d'Informatique Fondamentale de Lille
U.R.A. 369 du C.N.R.S., Université de LILLE I
Villeneuve d'Ascq FRANCE 59655
email: {rivals,delahaye,dauchet}@lifl.fr

O. Delgrange
Service Informatique
Université de Mons-Hainaut
Mons BELGIQUE 7000
email: delgrang@lifl.fr

Résumé

Nous présentons un algorithme de compression à dictionnaire dédié aux séquences d'ADN. Cet algorithme se déroule en deux phases. Durant la phase de factorisation, il construit l'arbre des suffixes du texte, pour sélectionner les répétitions du dictionnaire. Dans la phase d'encodage, les répétitions pour lesquelles une garantie de gain est établie, sont codées en binaire. Nous démontrons un théorème qui garantit la compression et montrons quelques comparaisons avec des compresseurs classiques. Les résultats complets sont disponibles par FTP anonyme à *ftp.lifl.fr:/pub/BIC/biologie/Cfact*. Ces expériences montrent que les textes d'ADN requièrent des schémas de compression spécifiques et soulignent l'utilité de notre méthode pour la classification de séquences en biologie.

Abstract

We present a text compression scheme dedicated to DNA sequences. This algorithm has two computation phases. In the parsing phase, the suffix tree is built to select repeats for the dictionary. In the encoding phase, selected repetitions for which a guarantee of gain is established, are encoded. We prove a theorem that guarantees the compression gain and report some comparisons with classical compression schemes. Complete results are available by anonymous FTP at *ftp.lifl.fr:/pub/BIC/biologie/Cfact*. These experiments establish that DNA sequences require special compression methods and show our algorithm utility for classification purposes in biology.

DNA sequences are texts over a 4 letters alphabet {A,C,G,T}. The exponential growing of the number of sequences and the recent apparition of whole chromosome sequences, create a huge sequence analysis work to achieve for their understanding. DNA contain numerous repeats. Most of the time, their biological function is unknown. Neither the repartition of sequences repeats is clearly established, nor their quantitative importance precisely assessed. Aside to well-known methods for identifying local similarities between given sequences ([SK83]), there is a real need for global comparison methods. A specific need emerges for methods that perform sequences classification upon various criteria, one of which is the sequence repetitiveness (cf. [HD94]). Various mechanisms such as DNA segments transpositions, inequal cross-over or defective DNA replication generate duplications of DNA contigs. Those exact repeats undergo elementary mutations mostly during DNA replication in successive cell divisions, and then become approximate repeats.

Some DNA repeats:

*Partially supported by the "GDR 1029 Informatique et Génomes" and by the "Groupement de Recherches et d'Études sur les Génomes"

- either are *"random phenomena"*, i.e. are simply due to the small size of the alphabet, which would constrain any randomly generated sequence to include such duplications,
- or proceed from real sequence evolution mechanisms which indeed duplicate DNA segments. Those repeats are *significative* from the information theory view-point.

A good lossless compression scheme is able to distinguish between random and significative repeats. Theoretical bases for this statement are found in Kolmogorov complexity theory, which is in fact an abstract theory of compression [CT91, Del94, LV93]. Practical applications of sequence analysis by compression algorithms appear in [Mil93, Riv94, RDDD94] and has been developed towards complete biological applications in [RDDD95].

In practice classical dictionary compression schemes perform really well on usual texts, but are unable to achieve compression on repetitive DNA sequences (see experiments results in section 5). A first attempt to build a compression scheme dedicated to genetic sequences is due to [GT93]: their algorithm is able to encode direct repeats and genetic palindromes (a special kind of symmetry between words), but is derived from a LZR scheme[1], i.e. a LZ77 scheme which avoids the limitation on the window size (cf. [ZL77, RPE81]).

The main principle of dictionary methods is to substitute an occurrence of a repeated factor by a pointer that references an earlier occurrence of the same factor. We define the encoding gain of a factor occurrence as the difference *in bits* between:

1. the original size of the factor,
2. the size of the whole replacement code for the occurrence, which includes the reference pointer and informations to locate the occurrence in the text.

We must point out that this is a "local" gain: this implies that the parsing strategy influences the local gain function. The overall gain on the sequence is the sum of the local gains.

Our aim is to conceive an optimized compression scheme which still substitutes repeats by codes with pointers, but does not fit in the framework of LZ77 or LZ78 because of their inadequation to biological texts ([ZL77, ZL78]). In this paper, we propose a guaranteed compression scheme thatselects repeats for its dictionary, in decreasing order of occurrences local gain, and always verifies that the encoding of a chosen occurrence yields a positive gain. Theorem 1 states that the compression is guaranteed, if sufficiently long repeats occur in the sequence to compress. Therefore, our algorithm *Cfact* is able to separate sequences that do contain significant repeats from those that do not, and also to classify compressed sequences according to their repetitiveness.

The following section gives an overall presentation of *Cfact*, section 2 details the coding of a sequence and section 3 explains the parsing of *Cfact*. Last section gives its complexity and lists the novelties of our algorithm, especially the guarantee theorem.

1 Overall presentation of *Cfact*.

Our purpose is to design new parsing strategy and coding scheme which make our algorithm suitable for repetitive genetic sequences. Repeated factors in ADN may be very long and may have occurrences far apart from each other. Therefore the window limitation prevents LZ77 schemes from encoding those repeats, while the parsing strategy of LZ78 schemes forces it to fill its dictionary, with small repeats that are due to the alphabet size.

Cfact executes sequentially the parsing phase and the encoding phase. The result of the parsing procedure is a chained list of sequence segments that have been chosen to be encoded ; it is denoted **LEZ: List of Encoded Zones**. We call **zones** those segments of the text, which are elements of LEZ. If a factor occurrence is selected by the parsing procedure, we say it becomes a zone and a new corresponding element is inserted in LEZ. In LEZ, zones are ordered according to their left to right position in the sequence, so the encoding can also proceed from left to right, while reading LEZ.

To be sure to choose the most significant repeats, parsing uses a suffix tree data structure to obtain the longest repeated factors with their proper list of occurrences. It selects interesting occurrences of those factors and creates for each of them an element in LEZ. Except non selected occurrences, the parsing classifies other factor occurrences in three types:

[1] We use the convenient notations of the compression schemes from [BCW90]

Reference Occurrence: the leftmost occurrence of the factor in the text is never encoded, it would be referred to, if another occurrence is encoded ;

2nd Occurrence: the second occurrence is the first occurrence selected by parsing, on the right of the reference occurrence. It is replaced by a code that contains a direct reference pointer towards reference occurrence ;

nth Occurrence (always with $n \geq 2$): all other selected occurrences on the right of the 2nd occurrence, their replacement code includes an indirect reference pointer, which is an index in an adaptive dictionary.

The special treatment for 2nd occurrences allows us to save the encoding of one index.

1.1 Use of the Suffix Tree.

The suffix tree is a well-known data structure that represents all factors of a text (cf. [CR94]), it has already been used in text compression in [FG89] but with a scheme derived from LZ78. The suffix tree is a tree in which a node:

- has a degree different of 1 (the node degree gives its number of childs),
- stores a factor of the text by a couple of integers giving its beginning position and its length.

On one hand if a node is a leaf, it represents a unique suffix of the text, on the other hand, if a node has degree $k > 0$, its factor is the *longest common prefix* of the factors of its k childs. In this way, it is maximal ; this is the reason why we call the factor stored in an internal node: **a maximal repeat**. The previous relation between a node and its childs implies two properties:

- the set of all occurrences positions of a factor is given by the set of all leaves beginning position in its subtree ;
- it is also the union of occurrences positions of its childs factors.

The ordered list of all positions of an internal node can be stored in an extra item of the node structure: we call it its **positions list**. From the preceding properties, one in-depth first exploration of the suffix tree suffices to build the positions list for all nodes and a list of all maximal repeats nodes. We denote this list: the **repeats list**.

The construction of the suffix tree for a text of length n takes only linear time, as well as its exploration. The suffix tree requires $O(n)$ memory spaces, while the size of the repeats list is in $O(n \log(n))$ space (cf. [CR94, Riv96]).

1.2 Example of compression.

We first give an example of a compression, which will be used to illustrate all phases of *Cfact*. In the following sequence s, the repeats list produced by the suffix tree exhibits the factors denoted by uppercase letters A, B, C. Their occurrences are numbered according to the text orientation, such that A_1 is the leftmost occurrence of A, and so on.

$$\mathrm{ttt}\overbrace{CAACGTC\rlap{$\underbrace{}_{C_1}$}TGT}^{A_1}GTACG\,\mathrm{tgactga}\overbrace{AATTCGC}^{B_1}\mathrm{cgcc}\overbrace{CAACGTCTGT}^{A_2}\mathrm{gccgc}$$

$$\underbrace{AATTCGC}_{B_2}\mathrm{aaa}\underbrace{CAACGTCTGT}_{A_3}\mathrm{atctta}\underbrace{TGTGTACG}_{C_2}\mathrm{attg}$$

A_1, B_1, C_1 are the respective reference occurrences for factors A, B and C. Note that C_1 overlaps A_1. A_2, B_2, A_3, C_2 are selected by parsing to be replaced by specific binary codes, which include a reference to the corresponding reference occurrence. This choice decomposes the sequence s, with A_2, B_2, A_3, C_2 being zones, as:

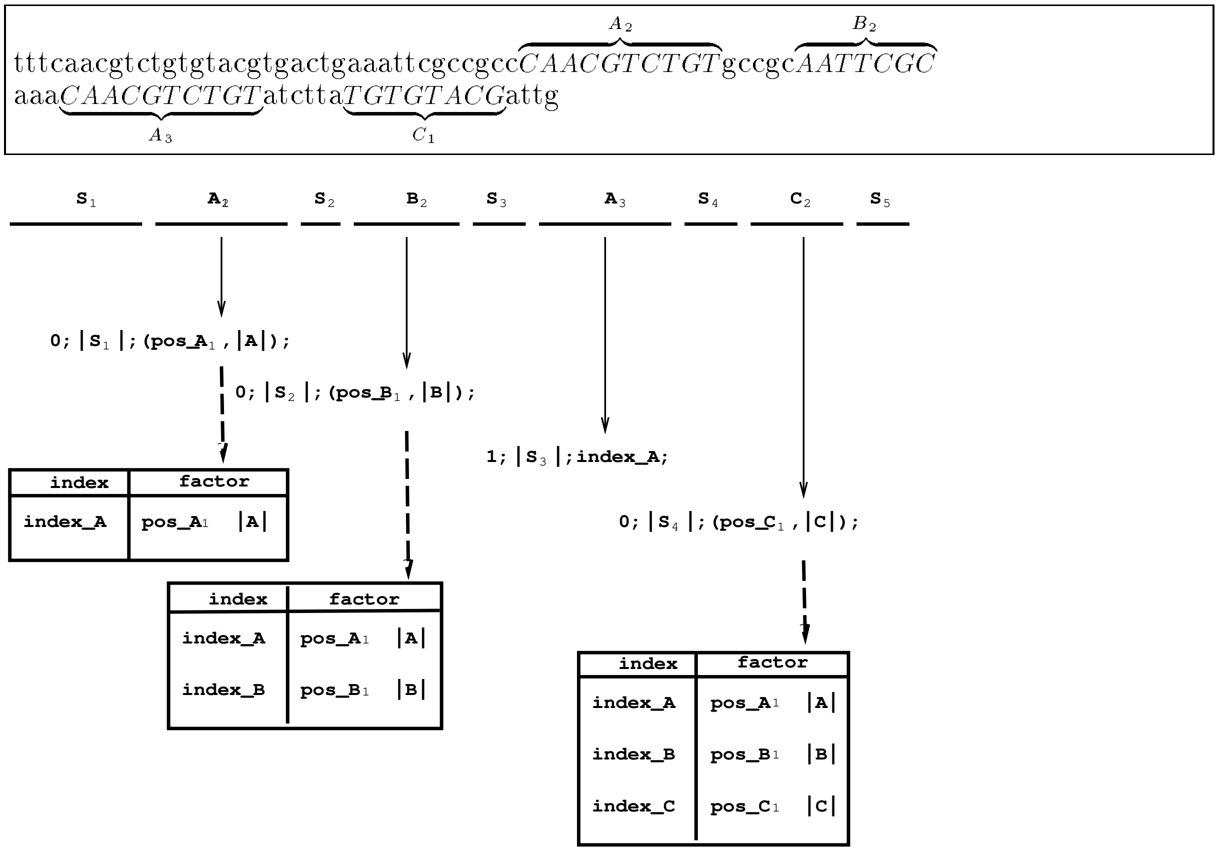

Figure 1: Encoding phase for the sequence s.

Code for	**Code value**		
number of encoded zones	4;		
	Zone type	**Offset**	**Pointer to earlier occurrence**
zone A_2	0;	$\|s_1\|$;	$(pos_{A_1}, \|A\|)$;
zone B_2	0;	$\|s_2\|$;	$(pos_{B_1}, \|B\|)$;
zone A_3	1;	$\|s_3\|$;	$(index_A)$;
zone C_2	0;	$\|s_4\|$;	$(pos_{C_1}, \|C\|)$;
remaining sequence	$Base2Binary(s_1.s_2.s_3.s_4.s_5)$		

Table 1: Complete code for sequence s.

A substituted occurrence (also called zone) may also overlap any reference occurrence. We remark that zones are separated by non-repetitive segments of the sequence. Let us denote them resp. by s_1, s_2, s_3, s_4, s_5 in the order of the sequence. Hence, the parsing gives the following decomposition of s:

$$s \quad = \quad s_1.A_2.s_2.B_2.s_3.A_3.s_4.C_2.s_5$$

s_1, s_2, s_3, s_4, s_5 segments are concatenated and all together encoded just by translating each base in a two bits code. This code is denoted by $Base2Binary$. The complete code for s is given in a readable format, in table 1. The length of a code or a string c is denoted by $|c|$.

The encoding phase for the zones is described in figure 1. The decomposition of s is shown in a sequence of horizontal lines. From left to right, the procedure associates a replacement code for each zone depending on the zone type. This code includes: a bit to differentiate between different types of zones, a relative position or offset to the previous zone end and a “pointer”. For 2nd occurrences, i.e. for A_2, B_2, C_2 this pointer is a couple of integers that gives the beginning position and the length of the reference occurrence. For nth occurrences, i.e. for A_3, it is an index in the adaptive dictionary, that only needs to be built during the decoding. Plain line arrows link a zone to its corresponding code. Pecked line arrows show the dictionary construction, which is a table that associates an index with a factor description.

2 Encoding scheme.

The coding of the compressed sequence is mainly illustrated in the previous example, but is detailed in the present section. First, the code of the sequence separates the binary code for all repetitive zones and what we call the *remaining sequence*. The non repetitive segments which appear between repetitive zones, are concatenated together and form the remaining sequence. The coding procedure does not try to encode this sequence in a special way because those segments seem incompressible for *Cfact*. It trivially encodes each base over two bits. In the example it is denoted by $Base2Binary(s_1.s_2.s_3.s_4.s_5)$.

We now describe the coding of the repetitive parts of the sequence. *Cfact* first encodes the number of zones given by the decomposition of the text, and then writes a binary code for each zone. As seen in section 1, there are two types of zone which must be identified by one bit: its value is “0” for 2nd occurrence zones and “1” otherwise. The zones are spread along the text and therefore the decoding process must be able to locate them inside the whole text. A code for a zone includes its relative sequence position, which is the difference between the end position of the previous zone and the beginning position of the current zone. For the example code of zone C_2, this item is $|s_4|$. Finally, there must be in the code, some informations that enable the decoding to find a previous occurrence, in order to recover the original sequence of the substituted current zone. Those informations differ depending on the zone type. Remind that the encoding is done from left to right. If it is a 2nd occurrence, a pointer which references the reference occurrence of the factor is encoded under the form of a couple of integers ; again for the example zone C_2, this couple is $(pos_{C_1}, |C|)$. As there is no restriction on how far back a pointer may reach, nor on how long a factor may be, the integers must be written in a self-delimited format. For integers, we always use the Fibonacci code which is proved to be asymptotically optimal (cf. [AF85]), but our algorithm works with any optimal code. Moreover, each time a 2nd occurrence is encoded, it is inserted in an adaptive dictionary and is associated with an index. For the encoding of a nth occurrence, the index can be used instead of the couple of integers ; for coding A_3, $index_A$ is output. This index is written in Fibonacci code.

The complete detail of a zone code is summarized in table 2. It also gives the length of each item. We use the following notations for the items of the current zone: P is its relative position, OR is the reference occurrence absolute position, L is the length of the repeated factor (the current zone is an occurrence of this factor), I the dictionary index of the factor. The Fibonacci code for a non-null integer item i is denoted by $Fibo(i)$.

	Type	Bit	Relative position	Reference pointer
Code value	2nd	0	$Fibo(P+1)$	$Fibo(OR+1).Fibo(L)$
	nth	1	$Fibo(P+1)$	$Fibo(I)$
Code length	2nd	0	$\|Fibo(P+1)\|$	$\|Fibo(OR+1)\| + \|Fibo(L)\|$
	nth	1	$\|Fibo(P+1)\|$	$\|Fibo(I)\|$

Table 2: Detail of a substituted zone code. Second part of the table give the length of each item.

3 Parsing strategy.

One original feature of *Cfact* is the separated execution of the parsing and the encoding. This separation allows the parsing to use better selection criteria than do usual dictionary compression schemes. A repeated segment of the text is chosen for future substitution if:

- it is a maximal repeat, i.e. if it belongs to the repeats list,

- it does not overlap an already encoded zone,
- its potential gain is positive.

The parsing goes through the repeats list given by the suffix tree, and for each repeated factor it simultaneously explores: (i) its ordered positions list and (ii) the LEZ. At a step in the exploration, the current element of LEZ is the nearest selected zone in terms of text positions, for the current occurrence of the factor. First occurrence plays the role of the reference occurrence and would not be encoded. For all other occurrences, the parsing checks that the current occurrence does not overlap an already selected zone, by looking at the current element in LEZ. If not, it evaluates the potential gain of the occurrence. In case the latter is positive, the current occurrence is selected and a corresponding zone is created in LEZ. In all other cases, the parsing skips to the next occurrence of the current factor.

3.1 Data structure for LEZ.

We detail the data structure for an element of LEZ, i.e. for a zone, in table 3. This helps to understand how a zone description is stored for later use in the encoding phase.

Item	Type	Semantic
Beg	integer	Zone beginning position in the text
End	integer	Zone ending position in the text
OccPos	integer	Absolute position of the first factor occurrence
Index	integer	Dictionary index for the factor (used for nth occurrence)
Next	zone pointer	Next element of LEZ

Table 3: Data structure of an element of LEZ, List of Encoded Zones.

First two items give segment position in the text, they are used to compute the offset relatively to the previous zone end. $OccPos$ gives the occurrence position of the factor first occurrence, it allows to encode the pointer if the current zone is of the 2nd occurrence. $Index$ plays the same role for nth occurrence, it stores the dictionary index associated with the factor when its 2nd occurrence was selected. $Next$ points to the following zone in LEZ.

3.2 Evaluation of an occurrence potential gain.

For each occurrence, the parsing procedure assesses its local gain as defined in section 1. It allows us to known if the encoding of an occurrence is a good step towards a better global compression. This evaluation consists in a comparison of the length of the segment being selected with the length of its replacement code. Exact computation of the latter requires to complete the parsing phase for each occurrence. Therefore, the length of the replacement code is only over-evaluated.

The calculus is done according to the coding strategy summarized in table 2 and depends on the zone type. The segment length is given by the factor node and for the replacement code, the formula sums the length of all code items as written is the second part of table 2. All items in the formula can be exactly assessed, except the offset of the current zone which is evaluated in the current state of LEZ. Until the end of the parsing phase, another zone can be added between the current zone and the previous zone, which would decrease the offset. Therefore, the offset is over-evaluated for all type of zone, and then **we always under-evaluate the occurrence gain**.

4 Complexity, advantages and guarantee.

The separation of parsing and encoding phases together with the evaluation of the local gain for each occurrence of a repeat, has a drawback on the time complexity of *Cfact*. For this section, we consider *Cfact* is applied to a text t of length n. The worst case quadratic complexity is not too bothering because the algorithm is not employed to save disk occupation or transmission time. Actually, low complexity is not a crucial requirement for sequences classification. Average computation time on a Sun-Sparcstation for a $50,000$ bases sequence is around 10 minutes, but finest optimizations are possible.

Proposition 1 The time complexity of *Cfact* is in $O(n^2)$ and its space complexity is in $O(n\log(n))$.

Proof The more complex phase in *Cfact* is the parsing. The repeats list contains at most n elements. For each of them the procedure explores simultaneously its positions list and LEZ, which can both contain n elements in the worst case. Therefore the time complexity is in $O(n^2)$. We saw in sub-section 1.1, that the largest data structure is the repeats list which takes $O(n\log(n))$. □

We prove a theorem that guarantees compression would occur if the sequence contains at least one "large" repeat. This property is an advantage of *Cfact* as a sequence classification tool. To our knowledge, no other method has this property.

Theorem 1 If the longest repeated factor of text t is of length l and occurs at least twice at positions i, j such that $i + l < j$, then the following sufficient condition:

$$2l - |Fibo(l)| \quad > \quad 3|Fibo(n)| + 1 \tag{1}$$

enforces the global compression gain of t to be positive.

Proof Equation 1 is a global over-evaluation of the minimal length of the longest repeated factor, such that the local gain of this repeat encoding is positive. This formula can be developed in a more precise form, if we replace the length of Fibonacci codes about which we know that, for an item i:

$$\left\lceil \frac{\log(i\sqrt{5}-1)}{\log(\frac{1+\sqrt{5}}{2})} \right\rceil \quad \leq \quad |Fibo(i)| - 1 \quad \leq \quad \left\lceil \frac{\log(1+i\sqrt{5})}{\log(\frac{1+\sqrt{5}}{2})} \right\rceil$$

The equation 1 constrains the repeat to be sufficiently long to generate a higher gain than the code of the number of encoded zones (this number is 4 in our example, see table 1).

The longest repeat of t is necessarily a maximal repeat and then is stored in first element of the repeats list. At this step, LEZ is empty, and the parsing finds that the occurrence in position j of f does not overlap any encoded zone. When it under-evaluates the potential gain of this occurrence, equation 1 forces the result to be positive. Then a zone in LEZ is created for this occurrence, and whatever new zones are selected afterwards, their local gain is under-evaluated and then positive. As the global gain is the sum of all positive local gains it is therefore positive. □

.

Cfact gathers a lot of interesting features to optimize the compression performance. We explain some of its novelties or advantages here under.

- There are no limitations on the size and location of an encoded repeated factor, then a pointer can encode any repeat in the text, wherever it is. Therefore the complete dictionary is:

  $$D = \{\text{all repeated factors in the text}\}$$

  It seems to be the most suitable option for genetic sequences.

- The encoding uses direct and indirect reference pointers. This empirically saves some bits because the couple (position, length) of the factor first occurrence has already been encoded earlier. Moreover, the significative repeats do not occur too often and the dictionary is quite small ; then an index in the dictionary costs less to code than a couple of arbitrarily long integers.

- All factors are examined in decreasing length, which means in decreasing potential gain, it avoids the parsing proceeding from left to right as in LZ78 or in [GT93].

- Each occurrence is selected and then encoded, only if it has a positive potential gain. A choice in the parsing is always a step towards better compression. This property allows us to obtain a guarantee compression theorem.

5 Conclusion.

In this section we report some experiments we performed to compare different compression schemes on real biological data. The main goal of these experiments is to compare *Cfact* to available compressors, and especially to confirm empirically *Cfact* guarantee of compression. We state some conclusions drawn from these experiences and give a sight on how *Cfact* can be used and extended.

5.1 Comparison experiments.

A dozen of compressors have been applied to sequences from different species and compared on their compression performance. The range of species varies from prokaryotic bacteria to higher mammalian species such as human species. It has been chosen to cover most of the species evolutionary tree. Long sequences have been prefered because of their likelihood (i) to include significative repeats, (ii) not to contain only genes. We report in table 4, the compression rates of 5 most interesting compression schemes, only on a few selected sequences. The complete set of results is available by FTP at *ftp.lifl.fr:/pub/BIC/biologie/Cfact* in flat file format. First three columns of the table give the sequence name, species of origin and length. Rightest three columns display the best compressors and compression rate.

Sequence	Organism	Length	*Cfact*	LZSS	LZW 15	Arith1	Arith2	Max %	Best
atatsgs	A. thaliana	9647	10.75	-31.19	-11.83	3.10	2.44	10.75	*Cfact*
atef1a23	A. thaliana	6022	20.77	-32.31	-14.85	1.30	0.30	20.77	*Cfact*
atrdnaf	A. thaliana	10014	9.30	-38.73	-15.00	0.82	-0.46	9.30	*Cfact*
atrdnai	A. thaliana	5287	26.60	-26.88	-11.97	1.19	0.28	26.60	*Cfact*
humghcsa	Human	66495	31.68	-35.93	-31.25	-16.05	-16.69	31.68	*Cfact*
hsg6pdgen	Human	52173	3.59	-32.42	-8.42	3.00	3.14	3.59	*Cfact*
xlxfg512	X. laevis	19338	25.49	-27.00	-4.19	3.03	3.86	25.49	*Cfact*
mmzp3g	M. musculus	10833	4.44	-38.17	-12.18	2.82	2.37	4.44	*Cfact*
celk07e12	C. elegans	58949	14.33	-21.62	-5.41	3.28	4.38	14.33	*Cfact*
ysch9177	S. cerevisiae	52664	2.02	-39.92	-11.13	2.26	2.11	2.26	Arith1

Table 4: Compression rates of five compressors on various DNA sequences.

Compression rate.

The compression rate does not take into account the trivial compression one may realize by translating each base of a sequence ASCII file in a two bits code: 75% of compression. The formula compares the sequence length as if each nucleotide takes 2 bits (and not one byte), to the bit length of the compressed sequence. It measures the size reduction between them: a 20% rate means that the compressed sequence size is 80% of the original size. For a sequence of 100 bases, the original length is 200 bits. Then if the compressed sequence measures 160 bits the compression rate equals 20%. Note that rates can be negative. At the end of a compression, a compressor can still compare the output version to the original sequence, and realize that it does not manage to compress the sequence. It could then output the original sequence preceded with a additional bit to announce this special case. Not all encoding schemes allows that, *Cfact* does not.

Selected compression schemes.

The compressors in competition are: *Cfact*; LZSS, a LZ77 scheme from [Bel86]; LZW15, a LZWelch scheme with 15 bits dictionary index (cf. [Wel84]) ; Arith1 and Arith2, two arithmetic encoders used with finite context models of order 1 and 2. All but *Cfact* are adapted from [Nel91]. We contacted the authors of [GT93] to compare with their algorithm, but it is no more available.

Observations.

The results shown here, have been selected to point out that repetitive sequences can be compressed, even if classical dictionary schemes do not manage to. The table confirms that *Cfact* is able to find and encode DNA repetitions. Many sequences especially from prokaryotic organisms give a null compression rate for *Cfact* showing that a majority of sequences are not repetitive ; none of those are shown here. Those organisms have smaller genomes than eukaryotes and generally optimize their use of the sequence, by overlapping genes, by “avoiding” introns and repeats. It shows *Cfact* can effectively been employed for sequences classification purposes.

The second observation is the unability of LZSS and LZW schemes to compress sequences even repetitive ones. In whole test over hundred of sequences, only one yields a positive compression rate. On the contrary, *Cfact* never gives a negative rate because of the evaluation of all occurrences local gain. Arithmetic encoders give low rates but are very constant, they always achieve compression thanks to known statistical biases in di- and trinucleotides usage.

5.2 Extension and future use of *Cfact*.

Cfact manages to detect and compress repetitive sequences. Its ability to separate sequences according to their repetitiveness can effectively be used for classification. At least two practical uses can be proposed. Firstly, a general classification of organisms according to their overall repetitiveness, especially those from which complete chromosomes are sequenced: the presence of repetition can be taken into account in phylogenetic problems as well as in the contig assembly problem. Secondly, a set of compressors including *Cfact* and arithmetic encoders, can be applied to all sequences at the entrance in a database, in order to annotate more precisely new sequences and to classify them.

The same compression scheme can be easily adapted to other regularities which occur in biological texts, such as genetic palindromes. The suffix tree allows us to identify genetic palindromes and repeats, therefore with little enhancements on the coding scheme, both regularities can be combined in one compression scheme. To implement these two compressors for palindromes, and for both regularities, is planned in our future work. Another extension is to design a compressor which encodes approximate repeats (that contains errors); but it requires a completely different approach to find and encode this sort of regularities. At present, *Cfact* can also be used to detect such repeats, because long approximate repeats would probably include some significative exact repeats. If a sequence can be compressed by *Cfact*, it would probably be interesting to try a compression scheme dedicated to approximate repeats.

References

[AF85] A. Apostolico and A.S. Fraenkel. Robust transmissions of unbounded strings using Fibonacci representations. Technical Report CS85-14, Dpt of Applied Mathematics, The Weizmann Institute of Science, Rehovot Israel, 1985.

[BCW90] Timothy C. Bell, John G. Cleary, and Ian H. Witten. *Text Compression.* Prentice Hall, 1990.

[Bel86] Thimoty C. Bell. Better OPM/L text compression. *IEEE Trans. Communications*, COM-34(12):1176–1182, December 1986.

[CR94] Maxime Crochemore and Wojciech Rytter. *Text Algorithms.* Oxford University Press, 1994.

[CT91] Tom M. Cover and Joy A. Thomas. *Information Theory.* Wiley Series in Telecommunications. A. Wiley-Interscience, 1991.

[Del94] Jean-Paul Delahaye. *Information, complexité et hasard.* Hermès, Paris, 1994.

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

[GT93] Stéphane Grumbach and Fariza Tahi. Compression of DNA Sequences. In *Data Compression Conference*, pages 340–350. IEEE Computer Society Press, 1993.

[HD94] Alain Hénaut and Antoine Danchin. Escherichia coli in silico. In F.C. Neidhart, editor, *Escherichia coli and Salmonella typhimurium cellular and molecular biology.* American Society for Microbiology, Washington DC, 1994.

[LV93] Ming Li and Paul M.B. Vitanyi. *Introduction to Kolmogorov Complexity.* Springer-Verlag, 1993.

[Mil93] Aleksandar Milosavljevič. Discovering Sequence Similarity by the Algorithmic Significance. In *Intelligent Systems for Molecular Biology*, pages 284–291. AAAI Press, 1993.

[Nel91] Mark Nelson. *The Data Compression Book.* M&T Publishing Inc., 1991.

[RDDD94] É. Rivals, O. Delgrange, M. Dauchet, and J-P. Delahaye. Compression and Sequence Comparison. In A. Apostolico, editor, *DIMACS Workshop on Sequence Comparison*, nov. 1994. Poster.

[RDDD95] É. Rivals, O. Delgrange, J-P. Delahaye, and M. Dauchet. A First Step Towards Chromosome Analysis by Compression Algorithms. In N.G. Bourbakis, editor, *First International IEEE Symposium on Intelligence in Neural and Biological Systems*, pages 233–239, Herndon (VA), USA, may 1995. IEEE Computer Society Press.

[Riv94] Éric Rivals. Compression pour l'analyse de séquences. In Maxime Crochemore, editor, *Actes de la rencontre pour les Recherches de Motifs dans les Séquences, GREG, Marseille fev-94*. Institut Gaspard Monge, Université de Marne La Vallée, 1994.

[Riv96] Éric Rivals. *Algorithmes de compression et applications à l'analyse de séquences génétiques*. PhD thesis, LIFL, Université des Sciences et Techniques de Lille, France, 1996.

[RPE81] M. Rodeh, V.R. Pratt, and S. Even. Linear algorithm for data compression via string matching. *Journal Association for Computing Machinery*, 28(1):16–24, January 1981.

[SK83] D. Sankoff and B. Kruskal. *Time Warps, String Edits and Macromolecules : the Theory and Practice of Sequence Comparison.* Addison-Wesley (Reading Massachusset), 1983.

[Wel84] T.A. Welch. A technique for high-performance data compression. *IEEE Computer*, 17(6):8–19, June 1984.

[ZL77] Jacob Ziv and Abraham Lempel. A Universal Algorithm for Sequential Data Compression. *IEEE Transactions on Information Theory*, 23:337–343, 1977.

[ZL78] Jacob Ziv and Abraham Lempel. Compression of Individual Sequence via Variable Length Coding. *IEEE Transactions on Information Theory*, 24:530–536, 1978.

Acknowledgments

We thank M-O. Delorme, E. Ollivier and A. Hénaut, the biologists from the Versailles University for numerous discussions and Olivier Baby for its essential help.