LOCATION OF REPETITIVE REGIONS IN SEQUENCES BY OPTIMIZING A COMPRESSION METHOD

O. DELGRANGE
Université de Mons-Hainaut
Avenue du champ de Mars 6, 7000 Mons, Belgique
Email: Olivier.Delgrange@umh.ac.be

M. DAUCHET
Laboratoire d'Informatique Fondamentale de Lille
CNRS URA 369, 59655 Villeneuve d'Ascq, France

É. RIVALS
Deutsches Krebsforschungzentrum (DKFZ)
Im Neuenheimer Feld, 280, 69120 Heidelberg, Germany

Suppose that a biologist wishes to study some local property P of genetic sequences. If he can design (with a computer scientist) an algorithm $\mathcal{C}$ which efficiently compresses parts of the sequence which satisfy P, then our algorithm TurboOptLift locates very quickly where property P occurs by chance on a sequence, and where it occurs as a result of a significant process. Under some conditions, the time complexity of TurboOptLift is $O(n \log n)$. We illustrate its use on the practical problem of locating approximate tandem repeats in DNA sequences.

1 Introduction

A compression method tries to reduce the size of a sequence (a text or a DNA sequence for example) by exploiting a property P of the sequence. The more relevant P is, the more compressed the sequence can be. For example, if the property P is "having duplicated regions", a DNA sequence consisting of two copies of the same segment can be compressed to nearly half its length. The compression gain measures the reduction in size. If a sequence cannot be compressed (i.e. the compression gain is less than or equal to 0), the property is irrelevant for the sequence.

The use of compression algorithms in the framework of pattern discovery works like hypothesis tests [1]: the hypothesis of the presence of a pattern is an "acceptable hypothesis" if the modelling of the data using this pattern leads to an effective compression. This principle is known as *inductive inference* and has its mathematical foundations from the works of Solomonoff [2], Kolmogorov [3], Chaitin [4] and Wallace and Boulton [5].

A sequence can be viewed as the description of an object (for example, the description of a DNA fragment) and each of its compressed versions is another

description of the same object. The *Kolmogorov Complexity* of a sequence is the length of its shortest description. If the length of a sequence is the length of its shortest description, it is incompressible. It has been proved [6] that a sequence is not random if it is compressible. The Kolmogorov Complexity is generally non-computable! [a] It is thus not possible to prove that a sequence is random. However, it is possible (but difficult) to prove that a sequence is not random: it is not random if we can compress it without prior knowledge. Practical compression algorithms are related to some specific properties and provide approximations of the Kolmogorov Complexity.

Common file compression programs perform poorly on DNA sequences and thus available prior knowledge or expertise in the biological domain is required [1]. For example, the use of approximate repeats in compression methods takes its inspiration from biological models of evolution.

There are some known compression methods for genetic sequences. In [7], a Lempel-Ziv like model is used to compute the *algorithmic significance* of a sequence. Such a method was also presented in [8] with the improvement of considering also inverted repeats. The authors of [9] extend a file compression algorithm to DNA by allowing mismatches to occur in "contexts". The paper [10] concerns the problem of finding the optimal compression of a sequence thanks to the encoding of its exact repeats. It present a heuristic approach to the problem. The paper [11] does not really present a compression method but it gives definitions of "compositional complexity" of strings. Recently, in [1], has been presented a loosely compression method based on the Lempel-Ziv model in which a repeated substring can be an approximate match of the original substring.

All these methods consider the compression gain as the global relevance of the exploited property. However, they do not help us to locate precisely, in the sequence, where the property is true and where it is not. In this paper, we present a new algorithm that does exactly that. The algorithm is called TurboOptLift. It takes, as input, the result of an already known compression method, applied to a sequence s, and provides a precise location of the regions of s where the property exploited by the method is true. **It is not a new compression method at all**! It requires the existence of a modular [b] compression method which exploits the property we want to study. It can be applied, for example to some of the above compression methods.

The algorithm proceeds by analyzing the result of the compression in order to locate the segments of s where the compression was not worthwhile. It is thus preferable to copy these segments as they are instead of compressing them.

[a] A compression algorithm cannot take into account all possible properties of the sequence.

[b] It means that some segments of the compressed sequence can be moved or deleted.

However, when a segment is copied, additional informations must be coded to allow the deciphering. Among other things, the length of the segment must be coded.

TurboOptLift solves the problem of finding the decomposition of s into "copy" segments and "compressed" segments, in order to **maximize the compression gain**. In "compressed" segments, the property is said to be relevant but in "copy" segments, it is said not to be so. This optimization problem is not obvious. Provided some conditions on the self-delimiting code *SD* used for the length of copied segments, TurboOptLift solves the problem in time $O(n\,|SD(n)|)$ where n is the length of the sequence. In practice, the code $Fibo$ [12] satisfies the conditions and thus the problem can be solved in time $O(n\,|Fibo(n)|) = O(n \log n)$.

At the end of the paper, we illustrate the use of TurboOptLift to locate *Approximate Tandem Repeats (ATRs)* **of a given motif** in DNA sequences. We also present the initial modular compression method which uses the *Wraparound Dynamic Programming (WDP)* technique [13]. Our algorithm selects ATRs that are significant at the scale of the whole sequence (an example is presented on the whole chromosome 11 of yeast). Moreover, the method seems able to **distinguish between random ATRs and ATRs generated by a specific process**.

The remaining of the paper is organized as follows. After preliminary notations, next section defines the notion of a compression curve and the new concepts of modular coding scheme. Section 3 shows how a compression curve can be improved and optimized if the compression method is modular. Rapid optimization of the compression is achieved by the algorithm TurboOptLift (section 5). Last section concerns the application to the identification of ATRs.

2 Preliminaries

We present here some basic definitions and properties concerning *sequences*, *compression methods* and *codes* (more details can be found in [14]). We introduce two new concepts: the modular coding scheme and the compression curve that are needed by our location algorithm.

Let $\mathcal{A}$ be a nonempty *alphabet*. This is a finite set of *letters* (or *symbols*). A *sequence* (or *word*) s over $\mathcal{A}$ is a finite sequence of symbols of $\mathcal{A}$. Its *length* is denoted $|s|$, it is the number of its symbols. The i^{th} symbol of s is denoted s_i, the *subword* (or *factor*) of s starting at position i and ending at position j is denoted $s_{i..j} = s_i s_{i+1} \ldots s_j$. A *prefix* u of s is a factor starting at the beginning of s: $u = s_{1..k}$ is the prefix of length k of s. The n^{th} power of s, noted s^n, is the word s concatened $n-1$ times to itself. In this paper, $\mathcal{B} = \{0, 1\}$ stands

for the *binary alphabet*, its symbols are called *bits*. $\mathcal{N} = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$ stands for the *alphabet of nucleotides*. For example, $s = \mathtt{AGACTGG}$ may represent a DNA sequence.

Given an input sequence s, a *lossless compression method* (more simply a *compressor*) $\mathcal{C}$ computes the *compressed sequence* $s' = \mathcal{C}(s)$ such that the entire sequence s can be reconstructed from s'. In other words, there exists a *decompression method* (or *decompressor*) $\mathcal{D}$ such that $s = \mathcal{D}(s')$. In practice, the *output alphabet* is $\mathcal{B}$.

Usually, a compressor achieves its work in two steps: the *analysis step* and the *coding step*. During the analysis step, informations are collected about the presence of a specific kind of pattern in the sequence. The goal of the coding step is the construction of the compressed sequence using a *coding scheme*, which is a set of rules for coding the input sequence using the patterns. It defines the syntax of a compressed sequence: a hypothetical compressed sequence s', written over $\mathcal{B}$, has a *valid syntax* for the compressor $\mathcal{C}$ if and only if it can be processed by the decompressor $\mathcal{D}$ without error.

The code enables to write items over $\mathcal{B}$. A *code* c must be injective to allow an unique deciphering. For example, we may want to code all letters of $\mathcal{N}$ or all integers of $\mathbb{N}$. The coding of an item using a code is called a *codeword*, this is a word over $\mathcal{B}$. For example, let us define the code NUC which maps a letter of $\mathcal{N}$ to a word over $\mathcal{B}$ such that $NUC(\mathtt{A}) = 00, NUC(\mathtt{C}) = 01$, $NUC(\mathtt{G}) = 10$ and $NUC(\mathtt{T}) = 11$. The sequence $s = \mathtt{AACGTAGGACT}$ can be coded as $NUC(s) = 00\,00\,01\,10\,11\,00\,10\,10\,00\,01\,11$.

A code is a *self-delimiting code* (often called *prefix code*) if no codeword is a prefix of another codeword. For example, NUC is a self-delimiting code. Self-delimiting codes are useful in the area of compression because when several codewords are concatened together, the obtained sequence is uniquely decipherable [6,14,12].

A compression method $\mathcal{C}$ has a *modular coding scheme* (more simply, $\mathcal{C}$ is a *modular compression method*) if each compressed sequence $s' = \mathcal{C}(s)$ can be decomposed into independent subwords which are the coding of corresponding subwords of the initial sequence s. That is to say

$$\begin{aligned} s &= (s_{1..i_1})(s_{i_1+1..i_2}) \dots (s_{i_k+1..n}) \\ s' &= I\, code(s_{1..i_1}) code(s_{i_1+1..i_2}) \dots code(s_{i_k+1..n}) \end{aligned}$$

for some positions $\{i_1, i_2, \dots i_k\}$. The prefix I of s' is the coding of the initial informations needed by the coding scheme. Each codeword $code(s_{i_j+1..i_{j+1}})$ is the subword of s' obtained by the coding of the corresponding subword $s_{i_j+1..i_{j+1}}$ of s. All such positions $\{i_1, i_2, \dots i_k\}$ are called *separating positions*.

Remark 1 *Although most of the practical compression methods (Lempel-Ziv, Huffman, ...) are modular, it is quite restrictive. It imposes to the sequel of* items *of the sequence to be well separated in the compressed sequence. This is not the case, for example, for the arithmetic coding.*
The *global compression gain*, defined as $g = |s| - |s'|$, is a measure of the reduction in size.
Remark 2 *In the case of DNA sequences, the formula is* $g = 2|s| - |s'|$ *because each letter of* $\mathcal{N}$ *must be coded over two bits before counting the number of symbols.*

For each separating position i_j, we define the *partial compression gain* $p(i_j)$ as the reduction in size obtained by $\mathcal{C}$ on the prefix $s_{1..i_j}$:

$$p(i_j) = 2i_j - |I\, code(s_{1..i_1}) code(s_{i_1+1..i_2}) \dots code(s_{i_{j-1}+1..i_j})|$$

By extension, $p(n) = g$ is the global compression gain and $p(0)$ represents the initial cost of the compression of s: $p(0) = -|I|$.

The *compression curve* of a modular compression method $\mathcal{C}$ applied to a sequence of length n is the partial curve defined for $0, i_1, i_2, \dots i_k, n$, which maps a position i to its partial compression gain $p(i)$.

Example 1 *Consider the 1000-bases long sequence of yeast chromosome 11 starting at position* 63700. *Suppose that there exists a modular compression method, called* $\mathcal{C}_{\texttt{TTC}}$*, which tries to compress a DNA sequence by exploiting the property of "being an* Approximate Tandem Repeat (ATR) *of* TTC [c] *" (such a method exists, it is presented in Sec.7). The compression curve of* $\mathcal{C}_{\texttt{TTC}}$*, applied to our 1000-bases long sequence is presented in Fig.1. The global compression gain is negative because this 1000-bases segment is not close enough to an* Exact Tandem Repeat (ETR) *of* TTC*. However a subword is very close to an ETR of* TTC*. It corresponds to the unique increasing segment of the compression curve. The other segments of the curve correspond to subwords that are not close to ETRs of* TTC.

3 Compression Curve Optimization

The goal of this section is to show how a modular compression method can be improved in order to maximize the global compression gain for a sequence.

An increasing segment of the compression curve exhibits a subword of the sequence for which the compression method is worthwhile: it produces a local gain. On the contrary, a decreasing segment corresponds to a subword which is lengthened by the compression method; the pattern does not occur frequently enough to produce a gain. It is of course preferable to copy the

[c] An ATR is a multiple copy, side by side, of the same motif with some mutations.

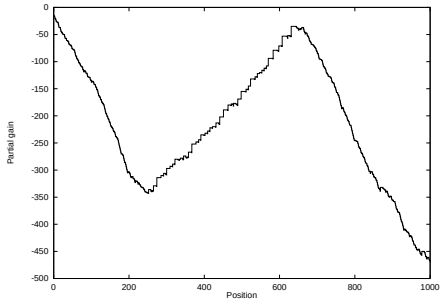

Figure 1: Compression curve of $\mathcal{C}_{\mathtt{TTC}}$ applied to the 1000-bases segment of Chr.11

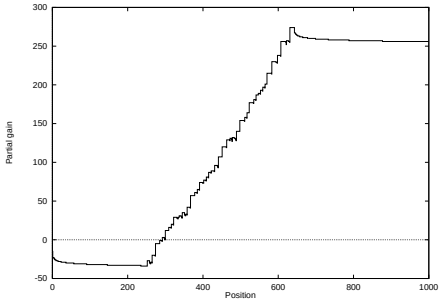

Figure 2: Compression curve of Fig.1 after the application of two liftings

corresponding subword of the sequence as it is instead of compressing it. To ensure that the resulting compressed sequence has still a valid syntax, we can only do such a copy between two separating positions i_j and i_k. In the compressed sequence, the encoding of the copy of the segment $s_{i_j+1..i_k}$ will be $[a_{\mathcal{R}}\, SD(i_k - i_j)\, NUC(s_{i_j+1..i_k})]$ with $a_{\mathcal{R}}$ being the *rupture flag*, telling to the decompressor that the usual coding scheme has been broken (we speak about a *rupture of the coding scheme*), and $SD(i_k - i_j)$ being the self-delimited encoding of the length $i_k - i_j$ of the segment. A few bits are then lost before the beginning of the factor copy.

We say that the rupture of the coding scheme induces a *lifting* of the compression curve because the right part of the curve is lifted up. If we apply this process on each decreasing segment of the curve of Fig.1, we obtain the curve showed in Fig.2. Each segment of the curve, corresponding to a copied subword, is replaced by a fixed *rupture curve*. It gives, for each position, the encoding cost of $a_{\mathcal{R}}$ and the length of the rupture if this position represented the end of the rupture.

Notice that, in this example, the choice of these two ruptures is optimal: the resulting improved compression curve has the maximal global compression gain.

Remark 3 *The rupture flag $a_{\mathcal{R}}$ must be a codeword unused by the coding scheme. Sometimes, the coding scheme must be adapted to provide such a subword. For example, for the Huffman coding, one of the codeword must be lengthened to create a new leaf in the Huffman tree. The initial compression method will suffer from this adaptation. The more shorter $a_{\mathcal{R}}$ is, the longer will be other codewords. This is thus a critical parameter. A choice must be made between favouring the initial compression method or favouring the application of ruptures.*

We can now make more precise the optimization problem we want to solve. Given:

1. a modular compression method $\mathcal{C}$ whose coding scheme provides a rupture flag $a_{\mathcal{R}}$;
2. an initial sequence s of length n that we compress using $\mathcal{C}$. The compression of s produces the compression curve as well the separating positions;
3. a fixed rupture curve which is completely specified by the length $|a_{\mathcal{R}}|$ of the rupture flag and the code *SD* used for rupture lengths;

we want to find the optimal decomposition of the sequence into subwords that must be compressed and subwords that must be copied as they are (*rupture subwords*) in order to maximize the global compression gain.

It is not an easy problem since the number of possible decompositions is an exponential expression of n.

4 ICL Codes and DCL Rupture Curves

In the preceding section, we mentioned that the code *SD* used for the rupture length must be self-delimiting. Moreover, we require that *SD* be *ICL (Increasing, Concave and Limited)*. We show how this property helps us to choose rapidly among several possible ruptures. We give an example of a self-delimiting ICL code: the *Fibonacci code*.

Let *SD* be a self-delimiting code which allows to write all integers over $\mathcal{B}$. It is *ICL* if it has the following three properties:

1. The length of the codewords is increasing: for all integers a, b with $a < b$, we have $|SD(a)| \leq |SD(b)|$.
2. The length of the codewords is concave: for all codeword lengths l_1, l_2, with $l_1 < l_2 : \#\{i : |SD(i)| = l_1\} \leq \#\{i : |SD(i)| = l_2\}$.
3. The increase of the codeword length is limited to 1 between two consecutive integers x and $x+1$: $|SD(x+1)| \leq |SD(x)| + 1$.

When the self-delimiting code used for the length of the ruptures is ICL, the associated rupture curve is *DCL (Decreasing, Concave and Limited)*. A DCL curve is a decreasing stair-like curve in which all "steps" are of height 1 except the first one which carries the cost of the coding of the rupture flag $a_{\mathcal{R}}$. Moreover, the "steps" of the curve are longer and longer.

The main interest of DCL rupture curves is that two curves do not have more than one crossing point (see Fig.3). Therefore, at the right of their crossing point, the one being below the other will never be the greatest one anymore.

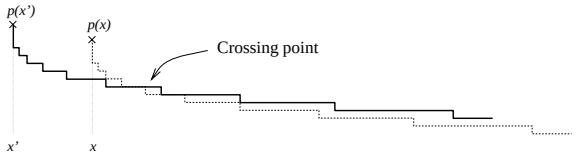

Figure 3: Two potential rupture curves that cross themselves

Because it is very difficult to satisfy the three ICL constraints at the same time, there are very few self-delimiting ICL codes. The *Fibonacci code*, defined in [12], is self-delimiting and ICL (the proof is in [15]). It codes integers using Fibonacci numbers. We denote by $Fibo(\ell)$ the Fibonacci coding of integer ℓ. It codes ℓ using $O(\log \ell)$ bits. In the remaining of the paper, we use $Fibo$ to code the length of a rupture [d].

5 The algorithm TurboOptLift

The goal of this section is to present the principle of our optimization algorithm and its time complexity. The algorithm is technical and the proofs are complicated. Interested readers may find more details in [15].

The TurboOptLift algorithm provides the unique optimal curve, minimal in ruptures among all optimal curves [e], in time $O(n \log n)$ where n is the length of the sequence.

To do this, the curve is processed from left to right. At step i, the curve is optimized over the interval $[0, i]$. Potential rupture curves whose have their starting position less than i are considered. Since the rupture curve is DCL, it is possible to prove [15] that the greatest rupture at position i can be selected in time $O(\log n)$. If this rupture improves the curve on $[0, i]$, it is applied. In fact, when a rupture is applied, the corresponding lifting is made formally: we only have to store the starting position and the length of the rupture. Thus, the application of a rupture is done in constant time. Since there are at most n steps (one step for each separating position), the time complexity of TurboOptLift is $O(n \log n)$.

Remark 4 *Miller, Myers [16] on one hand and Galil, Giancarlo [17] on the other hand have developed a dynamic programming algorithm to compute an alignment when the cost function is concave. The time complexity of their algorithm*

[d] We have developed another ICL self-delimiting code, called *PrefFibo*. It is moreover asymptotically optimal (see [15]) but its construction is very technical. Since $|Fibo(\ell)| \leq |PrefFibo(\ell)|$ for $\ell < 100000$, we do not consider it in this paper

[e] It is the only one for which the number of points that are on rupture curves is minimal.

is also $O(n \log n)$ *but the* log *operator comes from a dichotomical resolution of the problem. This is very different from our case where the* log *is related to the use of* $Fibo$ *to code the length of a rupture.*

6 Use of TurboOptLift Algorithm to Locate Repetitive Regions

It is now possible to use TurboOptLift as a "black box tool" to locate a specific kind of subword in DNA sequences. Such subwords are characterized as *repetitive* because they can only be located if they contain redundant informations that can be used to compress them.

We must have at our disposal a modular lossless compression method $\mathcal{C}$ which tries to compress a sequence s by exploiting this particular kind of repetitive subword. Thus **the choice of the compression method is very important**. Moreover, the compression scheme must provide a fixed rupture flag $a_{\mathcal{R}}$ (see Rem.3). If these two conditions are satisfied, the compression curve of $s' = \mathcal{C}(s)$ can be constructed. The length $|a_{\mathcal{R}}|$ of the rupture flag together with the code $Fibo$ used for the rupture length determine the DCL rupture curve. An example of an optimized curve is presented in Fig.4. In this exam-

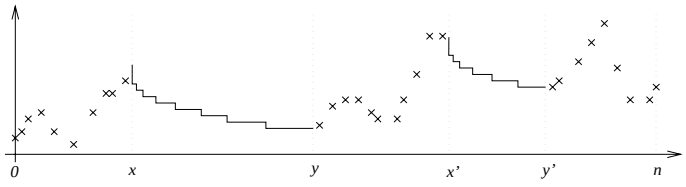

Figure 4: Example of an optimized compression curve

ple, three repetitive regions have been located: the subwords $s_{1..x-1}, s_{y+1..x'-1}$ and $s_{y'+1..n}$, and two non repetitive regions: $s_{x..y}$ and $s_{x'..y'}$.

The fact that TurboOptLift takes into account the cost $O(\log \ell)$ of a rupture of length ℓ is important because it makes the difference between significant repetitive subwords and the others: if a repetitive subword is not significant enough **at the scale of the whole sequence**, its local gain will be absorbed in a longer rupture and thus it will not be classified as repetitive. This is why the optimality of the solution is so important: all $O(\log \ell)$ items are taken into account for the optimal location.

Notice that after the optimization, we may apply another compression method to non repetitive regions. This enables the use of more than one compression method to make a classification between several types.

7 Location of Approximate Tandem Repeats in DNA Sequences

One of the mutational events of DNA sequences is the *tandem duplication*. It produces one or more copies, side by side, of a motif. For example, `ACT ACT ACT` is an *Exact Tandem Repeat (ETR)* of the motif `ACT`. Of course, because of additional mutations, tandem repeats may not be exact. We speak thus of *Approximate Tandem Repeats (ATR)*. For example, `AC GCT ACT ACT ATCT` is an ATR of `ACT`. ATRs are well studied because they are implicated in some human diseases and they may play a significant role in gene regulation. They belong to a bigger class of interesting repetitive DNA which is called the *dos*-DNA [18].

We use the TurboOptLift algorithm to locate ATR regions of a given motif m, of length p, in a DNA sequence s, of length n. Some preceding algorithms were dedicated to the location of ETRs or to the location of ATRs using heuristic methods [19,20,22]. There were also exact methods preceded by heuristics to eliminate a great amount of regions that could not contain any ATR[23]. Other methods are dedicated to the location of ATRs with a restrictive set of possible forms for each copy of the motif [20]. We present here an exact method which does not need any threshold value nor any restrictive definition of an ATR to be given. Of course, without restriction, every sequence is an ATR of every motif! The compression of s, exploiting the property "being an ATR of m", followed by the optimization using TurboOptLift will give us an optimal location (from our compression point of view), of the ATRs in s. However, the method requires the knowledge of the basic motif of the ATR and can only provide the ATR locations. It does not provide any model of the ATR regions like other methods do [23].

First, we need a modular compression method $\mathcal{C}_m$ which compresses s using the fact that it is an ATR of m. For this, we use the *Wraparound Dynamic Programming (WDP)* technique[24,13]. It computes the optimal alignment of the sequence s with the infinite ETR of m, i.e., m^∞. We use a simple cost function for the alignment: a penalty of 1 for any point mutation and a penalty of 0 for a match but it is possible to use other cost functions to tune the penalties given to the mutations. The time complexity of the alignment phase is $O(np)$.

Given this alignment, the compression and the construction of the curve are simple. We can see the alignment as a finite sequel of *elementary transformations* (matches or point mutations) which constructs s when m^∞ is known. The encoding of the compressed sequence is the encoding of the motif m followed by the encoding, from left to right, of the sequel of transformations:

$$Fibo(p-1)NUC(m)\,Fibo(l_1)t_1Fibo(l_2)t_2Fibo(l_3)t_3\ldots$$

where: - $Fibo(p-1)NUC(m)$ is the encoding of the motif m,
- $Fibo(l_1)$ is the encoding of the first l_1 consecutive matches,
- t_1 is the encoding of the first point mutation,
- $Fibo(l_2)$ is the encoding of l_2 consecutive matches, ...

We code $Fibo(0)$ if no match separates two consecutive point mutations. This coding scheme favours consecutive matches. We proved that each mutation $t_1, t_2, \ldots$ can be coded over 3 bits with one of the eight 3-bits codeword being unused (see [15]). This codeword can be used as the rupture flag $a_{\mathcal{R}}$.

The curve of example 1 was computed this way. After optimization, TurboOptLift provides the optimal curve showed in Fig.2. A 393-bases long ATR has been located.

Of course, the academic example presented here is obvious, we do not need the help of TurboOptLift to optimize the curve. On the other hand, if the whole sequence of chromosome 11 of yeast (666448 bases) is considered with the same motif `TTC`, the resulting compression curve is the one given in Fig.5. The details of the curve are completely invisible to the human eye.

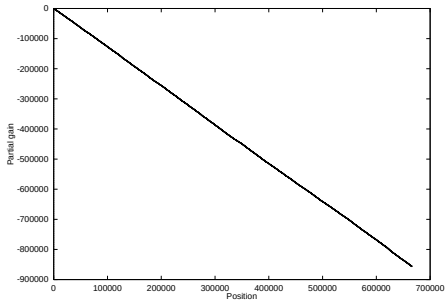

Figure 5: Compression curve for the whole chromosome 11 of yeast and the motif `TTC`

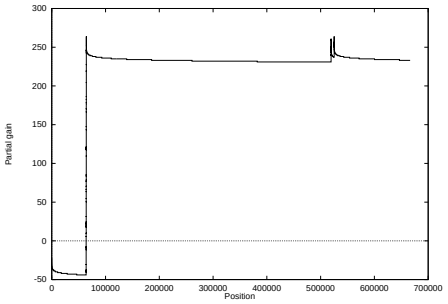

Figure 6: Curve of Fig.5 optimized

The optimization of the curve using TurboOptLift takes about 12 seconds on a Sparcstation 20 (96 MB main memory) and the resulting curve is the one given in Fig.6. Four rupture curves have been applied and three repetitive factors have been located (the three quasi-vertical wires of the curve). The first one is **exactly the same ATR as the one detected in the preceding 1000-base long window. Thus, this ATR is not only significant in the small window, but it is also significant at the scale of the whole chromosome**. The two other significant ATR regions are shorter (resp. 38 bases at position 519431 and 45 bases at position 525127).

A very important property of TurboOptLift is that **it seems able to make the difference between random repetitive regions and signif-**

icant repetitive regions [f]. We have applied the method to detect ATR regions in random sequences (of lengths 500, 1000 and 1000000) for every motif until length 5. Except in very few cases where ATR regions are shorter than 10 letters, **the method did not find any ATR!** Thus the ATRs detected in the chromosome do not seem to be caused by a random process.

References

1. L. Allison, T. Edgoose, and T.I. Dix. *Proc. ISMB*, pp. 8–16, Montreal 1998.
2. R. Solomonoff. *Inf. Control*, 7:1–22 and 224–254.
3. A.N. Kolmogorov. *Probl. Inf. Transmission*, 1(1):1–7.
4. G.J. Chaitin. *J. Assoc. Comp. Mach.*, 13(4):547–569.
5. C.S. Wallace, and D.M. Boulton. *Computer J.*, 11(2):185–194.
6. M. Li and P.M. Vitányi. Springer-Verlag, 2nd edition, 1997.
7. A. Milosavljevic and J. Jurka. *CABIOS*, 9(4):407–411, 1993.
8. S. Grumbach and F. Tahi. *Inform. Process. Management*, 1993.
9. D.M. Loewenstern and P.N. Yianilos. *Proc.IEEE Data Comp. Conf.*, 151–160.
10. É. Rivals, M. Dauchet, J-P. Delahaye, and O. Delgrange. *Proc. Genome Informatics Workshop*, Tokyo, 215–226, 1997.
11. J.C. Wotton. *in DNA and protein sequence analysis, Bishop M.J. and Rawlings C.J. editors*, 169–183, 1997.
12. A.Apostolico and A.S.Fraenkel. *IEEE Trans. Inform. Theory*, 33(2):238-245, 1987.
13. V. A. Fischetti, G. M. Landau, J. P. Schmidt, and P. H. Sellers. In *Proceedings of CPM*, 111–120, Tucson, 1992. Springer-Verlag.
14. J.A. Storer. Computer Sciences Press, 1988.
15. O. Delgrange. PhD thesis, UMH, 1997. Available at `http://sun1.umh.ac.be/~olivier/these.html`.
16. W. Miller and E. Myers. *Bull. Math. Bio.*, 50:97–120, 1988.
17. Z. Galil and R. Giancarlo. *Theor. Comp. Sci.*, 64:107–118, 1989.
18. R.D. Wells and R.R. Sinden. In K.E. Davies and S.T. Warren, editors, *Genome Analysis, 7*, Cold Spring Harbor Laboratory Press, 1993.
19. G. Benson and M.S. Waterman. *Nuc. Acids Res.*, 22(22):4828–4836, 1994.
20. É. Rivals, O. Delgrange, J.P. Delahaye, M. Dauchet, M.O. Delorme, A. Hénaut, and E. Ollivier. *CABIOS*, 13(2):131–136, 1997.
21. É. Rivals, M. Dauchet, J.P. Delahaye, and O. Delgrange. *Biochimie*, 78(4): 315–322, 1996.
22. G. Benson. In Waterman and Pevzner [25], 20–29.
23. M.-F. Sagot and E.W. Myers. In S. Istrail, M. Waterman and Pevzner [25].
24. E.W. Myers and W. Miller. *Bull. Math. Biol.*, 51:5-37, 1989.
25. S. Istrail, M. Waterman and P. Pevzner, ed. *RECOMB 98*. ACM Press, 1998.

[f] By definition of an arbitrarily long random sequence, every finite subword can appear.