# Fast Discerning Repeats in DNA Sequences with a Compression Algorithm

Eric Rivals [1] E.Rivals@dkfz-heidelberg.de
Max Dauchet [2] dauchet@lifl.fr
Jean-Paul Delahaye [2] delahaye@lifl.fr
Olivier Delgrange [3] olivier@sun1.umh.ac.be

[1] Deutsches Krebsforschungzentrum (DKFZ)
Im Neuenheimer Feld 280, Heidelberg 69120, Germany

[2] LIFL, URA 369 CNRS, Université Lille I
Villeneuve d'Ascq 59655, France

[3] Service Informatique, Université de Mons-Hainaut
Avenue Maitriau 15, Mons 7000, Belgium

**Abstract**

*Long direct repeats in genomes arise from molecular duplication mechanisms like retrotransposition, copy of genes, exon shuffling, ... Their study in a given sequence reveals its internal repeat structure as well as part of its evolutionary history. Moreover, detailed knowledge about the mechanisms can be gained from a systematic investigation of repeats. The problem of finding such repeats is viewed as an NP-complete problem of the optimal compression of a sequence thanks to the encoding of its exact repeats. The repeats chosen for compression must not overlap each other as do the repeats which result from molecular duplications. We present a new heuristic algorithm,* Search_Repeats, *where the selection of exact repeats is guided by two biologically sound criteria: their length and the absence of overlap between those repeats.* Search_Repeats *detects approximate repeats, as clusters of exact sub-repeats, and points out large insertions/deletions in them.* Search_Repeats *takes only 3 seconds of CPU time for the genome of* Haemophilus influenzae *on a Sun Ultrasparc workstation.*

## 1 Introduction

A class of evolutionary mechanisms duplicates an existing segment of DNA or RNA and reinserts a DNA copy in a longer DNA molecule, thereby creating a repeat. If the segment contains one or more complete genes, the duplication directly affects the content in gene. The new copy of the gene, which may not be necessary for the organism, may evolve further for instance by punctual mutations or exon shuffling, and thus acquire a new function. If the DNA piece does not include any gene, this evolution "by segment", as opposed to punctual mutations, nevertheless changes the linear structure of the target DNA molecule. It may break an existing gene at its point of insertion, modify the overall repartition of genes or the chromatin 3D structure. Those alterations of the genome are indirect, but crucial. Different copies of a repeat are a source of observation, measurement and comparison of further evolutions. For instance, different rates of punctual mutation, amplification of tandem repeats [21] or deletion of a segment can only be revealed if observed within a repeat[1].

**What are the characteristics of evolutionary repeats?**

Mechanisms like retrotransposition generate repeats up to a length of thousands nucleotides and copies can be inserted at positions arbitrarily far away from the origin. These are two important characteristics. Moreover, not only *real repeats*, i.e. molecularly duplicated segments, but also *random repeats*, repeats which arise for statistical grounds, are present in DNA. Indeed in DNA

[1] Here, "repeat" must be understood in a broad meaning: it can also be a piece of DNA which is found in the genomes of different species or individuals.

sequences, "random" short repeats must occur because the alphabet contains only four letters. The average size of such random repeats grows with the sequence length. If a repeat is too small, it becomes indistinguishable to see if its origin is random or molecular; thus in this work, the result of a molecular duplication is assumed to be a repeat of a certain length. To study molecular duplication in sequences, one must find a way to distinguish random from real repeats.

*Notations*: In the figures, a simple horizontal line represents a DNA sequence. Among the light grey segments that represent many occurrences of the same subsequence, the leftmost is named $A$ and the following ones $A'$, $A''$, ... (they are most often inside, but also above or below the line of the sequence.) An arrow represents the length of the segment above which it is drawn. In the text, $s$ denotes the input sequence over the alphabet $\mathcal{A}$, and $n$ its length.

Another important characteristic is the *absence of overlap*. A molecular duplication is a way to generate a segment of DNA. But a segment of a specified length at a given beginning position can only result from one duplication. In figure 1, the occurrences $A'$ and $B'$ overlap, the corresponding segment is decomposed in $C$, $A' \cap B'$ and $D$. Obviously, $A' \cap B'$ stems from the duplication of either $A$ or $B$, since only one but not both of those events may have happened.

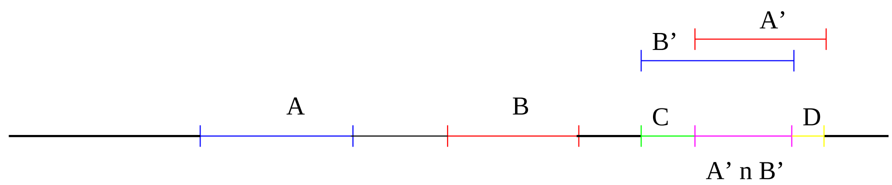


Figure 1: The segment $A' \cap B'$ results from the duplication of either $A$ or $B$, but not from both. One must choose between both duplications, the overlap must be avoided.

**Related works: How can we detect such duplications in a sequence?**

Looking for the approximate or exact occurrences of a given pattern in a text are both well-known problems in computer science. Such methods are unsuitable because patterns are not known in advance. Our problem requires first to discover the repeated patterns with their occurrences and second to choose a combination of repeats without overlap and containing a maximum number of real repeats. This combination would then suggest a molecular origin for as many as possible of the discovered repeats. Three ways of tackling this problem have already been studied:

• **Algorithms looking for optimal local alignments** serve for searching similar segments between a query sequence and all sequences in a database. With the query sequence as the database, such algorithms look for approximate repeats according to some distance or dissimilarity measure. Among others, the Smith and Waterman [19] algorithm computes exactly all optimal local alignments scoring greater than the input threshold score in $O(n^2)$. Numerous heuristic algorithms like BLASTN [2], FASTA [16] give approximate solution in the same worst case complexity, but operate much faster on complete databases. Nevertheless, the time requirements of those algorithms constitute a bottleneck. With an exact dynamic programming algorithm for ungapped alignments, Agarwal and States reports a 4 days computation on 6 workstations for a 3.6 Mb sequence of C.Elegans [1]. A search of the *Haemophilus influenzae* (1.8 Mb) with BLASTN with default parameters requires more than eight hours on a Sun Ultrasparc workstation.

• **Pattern discovery algorithms** search segments conserved among many sequences, i.e. approximate repeats which have been punctually mutated. We can divide them in two subclasses. The first encloses approximate methods. An example is the algorithm of Leung and al. [13] which finds approximate repeats as a sequence of exact blocks matches separated by mutated blocks. Many parameters allow to tune up the results. In the other subgroup, the exact methods are constrained to perform an exhaustive search among all putative patterns whose number grows exponentially on the length of their length. This implies high complexities and those algorithms are thus used in practice to search for short weak patterns in many short sequences (see [6] for a survey.)

Both previous classes of algorithms score their alignments by summing the individual scores of all aligned residues-pairs. Only punctual differences are taken into consideration. If two sequences share a similar part and one has undergone a segment deletion in this part, then such algorithms report not one alignment showing the similar part, but two shorter local alignements (see figure 2.)

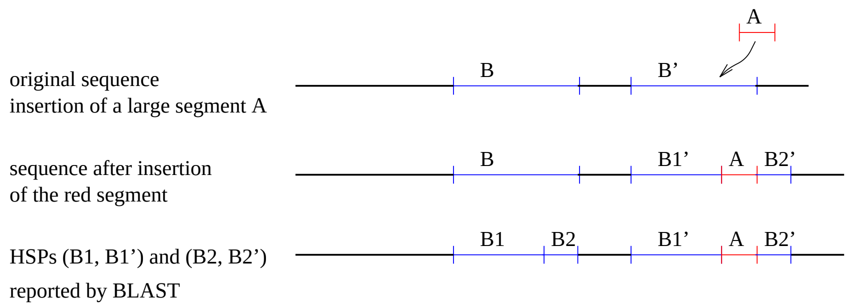


Figure 2: A duplication altered by the insertion of a segment. The original sequence (first line) undergoes the insertion of $A$ inside the repeat-occurrence $B'$ (result on the second line.) An algorithm like BLASTN reports 2 different alignments (called HSP for High Scoring Pairs), $(B_1, B_1^{'})$ and $(B_2, B_2^{'})$, instead of one $(B, B')$.

• **Compression methods**: Lossless compression of a text is achieved by replacing exact copies of sub-words by an encoded pointer that indicates the location of the first occurrence of this sub-word. These algorithms not only detect repeats but also choose a decomposition of the text in a suite of factors[2], some of which are repeats and therefore replaced by a pointer. The decomposition or factorization is the result of the parsing phase (an example is shown in figure 5.) In comparison to other methods, its computation requires one more step to select non overlapping repeats instead of reporting all of them. This selection may reduce the output, making it more usable and legible.

In previous application to genetic data, for instance in Biocompress [11] or in the work of Milosavljević and Jurka [15], parsing is done in the spirit of the LZ78 compressor: on-line and left-to-right. The constraint of being on-line comes from the stringent speed requirements imposed in "classical" uses of text compression (in a program like gzip under Unix.) It has no reason to be in our context. Moreover, it may results in an unsuitable factorization of the repeats in a DNA sequence (see figure 3.)

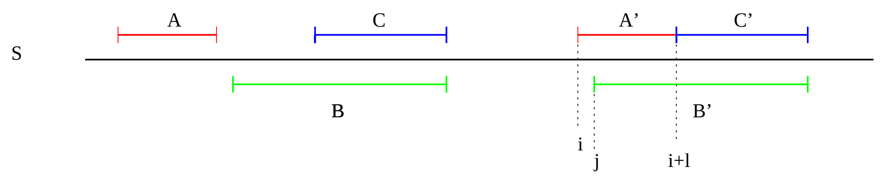


Figure 3: Unsuitable factorization of the sequence by an on-line left-to-right parsing. When the parsing reaches the position $i$, it finds $A'$ as the longest repeat beginning at that position; it factorizes it and reaches position $i + l$. Then, in the same manner it factorizes $C'$. But it misses a much larger duplication: the one of $B'$ which would explain the origin of the whole segment by only one evolutionary event.

Another advantage of compression is that trying to compress a sequence is a *sufficient test of non-randomness* (cf. [14] chap. 5 p. 377.) The outcome of a "compression test" is positive, if the encoded sequence is shorter than the original sequence (a fair comparison must be done between

[2]We use the word "suite" instead of "sequence" when it refers to an ordered list of items, i.e. to sequence which is not necessarily genetic. Here it is a list of factors ordered on their position in the text.

the lengths in bits), and negative otherwise. As it answers a *sufficient test*, a positive outcome means that the repeats found in the factorization are not random "structures". Indeed, this is because random sequences are incompressible [14]. On the other hand, it cannot be concluded that a sequence is random in case of a negative outcome. This is valid for any type of compression[3].

In this work, we chose the compression approach because of the last two mentioned advantages. The problem of detecting repeats in a sequence is formalized as the problem of optimizing the compression of the sequence, when only encoding duplicated segments is allowed. It is known that computing the optimal compression is an NP complete question. We propose a new compression algorithm based on an off-line heuristic parsing procedure. Exact repeats are examined in decreasing length order and a copy of repeat is chosen for the factorization if it avoids overlapping with factors that already belong the factorization.

The next section shows how a repeat is encoded, which allows a precise formalization of the problem of finding the optimal compression. In section 3, the algorithm of the parsing procedure, which uses a suffix tree of the sequence, is detailed. The time ($O(n^2)$) and space ($O(n)$) complexities are stated. In the last section, we report on the execution time and some results from the study *Haemophilus influenzae* genome [10]. This section also illustrates why a positive compression outcome assesses the significance of the repeats found.

# 2 Coding of repeats and optimal compression.

In this section, the problem of optimal compression by encoding of repeats is precisely presented and defined. Its NP completeness is also stated. Though we avoid technical details, sufficient precisions about the coding is given in order to explain the heuristic of our parsing procedure.

We use compression as a way to detect if some repeats in the sequence are significant. The answer is given by the comparison of the lengths in bits of the encoded and original description of the sequence. As DNA is built with $4 (= 2^2)$ possibles bases, each of them might be encoded over 2 (the exponent) bits. Therefore when measured in bits, the original DNA sequence's length equals twice its length in bases. Any encoded version of a sequence is produced in a binary format. This allows to compare the length of both descriptions, and implies the following formulae for the compression gain: gain = original size − encoded size, and rate: rate = $1 - \frac{\text{gain}}{\text{original size}}$

Before looking at the overall problem, we have to describe i) how a single repeat can be and is encoded, ii) how a combination of several repeats is encoded, i.e. how the encoded sequence is written. From now on, some definitions are required. A *string or word* $s$ is a suite of letters taken from an alphabet $\mathcal{A}$, e.g. a DNA sequence is a string over the alphabet $\{A, C, G, T\}$. A *substring* of $s$ is a suite of consecutive letters that occurs in $s$, while a *factor* is a positioned occurrence of a substring in $s$. A *repeat* of $s$ is a substring which has several different occurrences in $s$; its length is the substring's length. A *code* is a string over the binary alphabet. $|x|$ denotes the length of $x$ if $x$ is a string and the cardinal of $x$ if it is a set.

## 2.1 Encoding of a single repeat.

A repeat may have several occurrences, but for the sake of clarity we consider in this sub-section one with only two occurrences. We mentioned that the encoding of a repeat is achieved by replacing its rightmost occurrence by a pointer to its leftmost one. The latter is called the *source occurrence* and the former the *target occurrence*. Two important issues are: What informations must be enclosed in this pointer so that the decompression algorithm can rebuild the original sequence from the encoded sequence? How is each item of information precisely coded?

[3] One might be surprised because negative outcomes are rare in the usual compression of text or program files on a computer disk. But this is not a contradiction, it just "tells", what is obvious, that the files compressed are not random. Simply, the algorithms used in those cases are specialized for those types of texts. They "know" the characteristic structures that makes them non random, they look for them and achieved compression.

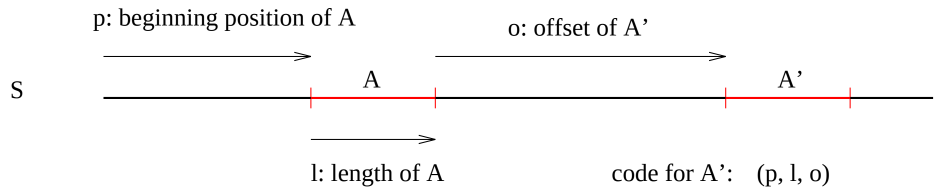


Figure 4: Encoding of an occurrence of a repeat.

As shown in figure 4, the code for the target occurrence $A'$ is a triplet $(p, l, o)$ formed by: $p$ the beginning position of $A$, $l$ the length of the repeat and $o$ the offset of the target occurrence. Each of these are integer items. To encode those integers, we chose to use the Fibonacci code[4] [3].

During decompression, the piece of code for $l$ must be read and distinguished from those for $p$ and $o$. In other words, when reading the suite of 0 and 1 from the encoded sequence, the decompression algorithm must delimit the code for each item. The Fibonacci code fulfills this constraint because it is self-delimiting; moreover, it is also one of the most economic codes to have this property [9].

Characteristics of DNA repeats imply that all these items may take their value in a very broad subset of the integers. Clearly, items $p, l$ and $o$ could be nearly as large as the sequence itself. As we do not know an a priori distribution of those values, nor do we know the length of the sequence in advance, our integer encoding should encode any integer. Thus, a fixed-size code cannot be used, and with a variable-size code it is logical to assign short codes to small integers. The length of the Fibonacci code grows logarithmically in function of the integer to encode. As the items to encode do not exceed $n$, the size of a pointer code is in $O(\log(n))$.

## 2.2 Encoding of a combination of repeats.

A combination of repeats or *factorization* of the sequence is a set of factors which are all a target occurrence of some repeat and do not overlap with each other. For each target, the corresponding source occurrence is known. Note that there is no constraint on the overlapping of sources with themselves nor on sources with targets[5].

Figure 5 shows an example of a sequence encoding: the first line displays the source occurrences of the repeats, the second shows the factorization with the target occurrences and the third one gives the code (in a non binary representation) and its meaning. Portions of the sequence which are not covered by a target (the black segments denoted $d, e, f, g, h$) are concatenated in their order of appearance and form the *remaining sequence*. The latter is encoded by replacing each base by its two bits code. The binary encoding of the whole sequence is made of two distinguishable parts: the code of the ordered list of pointers (one pointer per target in the factorization) and the code of the remaining sequence[6]. So the "pointer part" is simply the list of the code for each pointer in left-to-right order, preceded by the Fibonacci code of the number of pointers (this serves as the self-delimitation of the list.) A target factor which is coded by a pointer as shown in section 2.1 is also called an *encoded zone*.

**Gain of an encoded zone.**

We defined earlier the compression gain of the sequence. Now the *gain of a single encoded zone* also has a meaning: it is the number of bits saved by the encoding of the target. I.e., the difference of the lengths in bits between i) the code of the factor in the original description (2 bits per bases) $2 * l$ and ii) the code of the pointer that replaces it: $|code(p, l, o)|$ (we neglect the

[4] The Fibonacci code takes an integer as input and produce a binary representation of it. Instead of the powers of two in the natural binary representation, the Fibonacci numbers are used as "base" to write the binary encoding.

[5] When a target overlaps a source, then the occurrence denotes a tandem repeat [8].

[6] The separation of the two parts in the code is just a matter of "format". As each part and each item in a part must be delimited, it makes no difference if they are mixed together or well separated. This format allows to make further experiments with the remaining sequence alone.

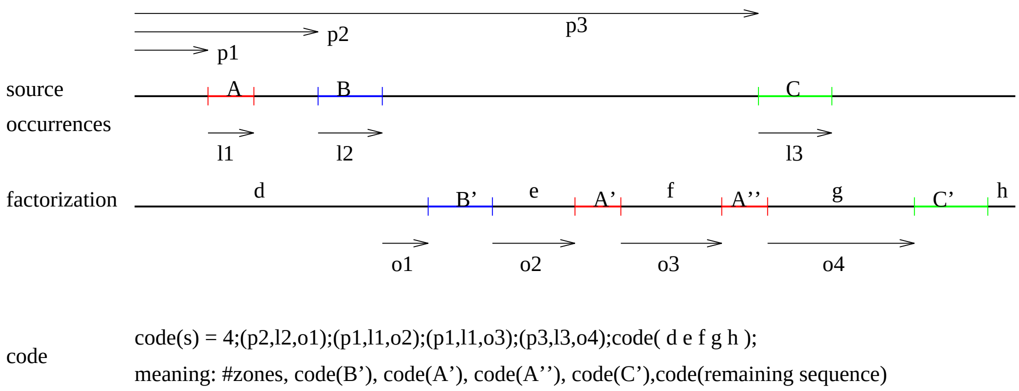


Figure 5: Encoding of a factorized sequence.

increase in the number of targets in the factorization.) We see that the gain of a zone is of the form $O(l) - O(log(n))$. Then the length of the factor is the principal parameter that governs the gain for a zone and thereby the overall gain for the whole sequence.

### 2.3 Optimal compression.

We have described a coding scheme which takes as input a sequence with a factorization of it and produces a binary code for it. The problem of the optimal compression of a sequence is now well defined and consists in finding the factorization which yields the shortest coded sequence. Many factorizations may give an optimal code for the sequence. For a given sequence and a given integer $K$, determining if the shortest encoded sequence is shorter than $K$ is known to be NP-complete for a coding scheme like ours (cf. theorem 2.3.5, chap. 5 in [20].)

## 3 Heuristic factorization algorithm.

As computing an optimal factorization of the sequence is an NP-complete problem, we present an heuristic factorization algorithm. The choice of the repeats aims at maximizing the compression gain (of a zone) and is based on two genetically sound criteria: the length of the repeats and the absence of overlap between those repeats. Here follows a description of the parsing algorithm. It takes as parameter the minimal length (denoted min_length) of the repeats which are considered; a value below $\log(n)$ is not recommended (see "Compression gain of a zone" in section 2.2.) It returns a factorization as an ordered list of target-factors. The suffix-tree data structure [8], which is used in the parsing, is presented in appendix A.

The main idea of the algorithm is the following. All repeats exceeding min_length (from now on we write only "repeat") are examined in decreasing length order. The algorithm considers each putative target occurrence of the current repeat. It selects one if its encoding does not conflict with the encoding of an occurrence of a better (longer) repeat, i.e. if it does not overlap a target occurrence which is already in the factorization.

We proceed as follows. First, we build in $O(n)$ time a suffix tree of the input sequence; it requires $O(n)$ space. From it, we compute the ordered list of repeats $L_R$: the repeats corresponding to the factors of internal nodes are inserted in $L_R$. These have two properties: i) the factor of a node is shorter (indeed a prefix) than the ones of its children, thus in $L_R$ the children always precede their father; ii) a factor $x$ of a node is "maximal"[7]: all its prefixes which are longer than its father's factor occur at the same starting positions than $x$, they are thus never more interesting

[7] The meaning of "maximal" is defined in the appendix.

to encode than $x$ (ex in figure 7: "caga" is maximal, but "cag" is not.) This justifies our choice for $L_R$.

The main loop goes through $L_R$. For any repeat, the list of all its occurrences, denoted $l_{occ}$, can be obtained by reading the leaves of the subtree rooted by its corresponding internal node. Assume that once this list is built, it is stored in the internal node; then we compute the list of the current node by merging the ones of its children.

All factors already inserted in the factorization are kept in another ordered list, denoted by $\mathcal{F}$, for factorization. At completion, $\mathcal{F}$ is the result of the procedure. The factors in $\mathcal{F}$ are ordered according to their beginning position in the sequence. Thus, checking the overlap of all occurrences of the current repeat with all zones in $\mathcal{F}$ can be done in $O(n)$ (because both lists are ordered.)

Another precaution avoids considering positions where a longer factor has already been encoded. Once an occurrence is put in $\mathcal{F}$, it is deleted from $l_{occ}$ which will be merged to build its father's positions list. Indeed, an occurrence of a repeat considered later in the course of the parsing and at the same position can only be shorter and would then overlap the current target-factor.

```
PARSING(SEQUENCE, MIN_LENGTH)

Begin
    T = compute_ST(sequence)
    L_R = sort(get_list_maximal_repeats(T, min_length))
    \\initialize F
    F = empty list
    \\main loop: examine all repeats in L_R
    For all r ∈ L_R Do
        \\get the list of positions of occurrences of r
        node(r) → l_occ = compute_list_occurrences(node(r))
        curr_zone = first_elem(F)
        \\init curr_occ to the second occurrence, the first being the source one
        curr_occ = next(first_elem(l_occ))
        While ( curr_occ ≠ nil ) Do
            curr_zone = get_closest_zone( curr_zone, curr_occ )
            If ( overlap( curr_zone, curr_occ ) ) Then
                curr_occ = next(curr_occ)
            Else
                \\put curr_occ in F insert(F, curr_occ)
                tmp = previous(curr_occ)
                delete(l_occ, curr_occ)
                curr_occ = next(tmp)
            EndIf
        Done
    Done
    \\return the computed factorization
    return F
End PARSING.
```

Figure 6: Algorithm of the Parsing procedure.

A formal description of the *Parsing* procedure is given in figure 6. The call of *get_closest_zone* moves *curr_zone* towards the zone which is the closest to *curr_occ* in the sequence. Then *overlap* computes if there is an overlap between those two factors.

**Proposition 1** If $n$ denotes the length of the input sequence, the worst-case space and time complexities of *Search_Repeats* are respectively in $O(n)$ and $O(n^2)$.

**Proof** Let us look at each data structure:

1. The input sequence and its suffix tree $T$ both take linear space [7].
2. $L_R$ does not contain all internal nodes of $T$ whose number is necessarily less than $n$ [7].
3. When a repeat $r$ is examined, its list of occurrences positions $l_{occ}$ is stored in its node. But when its father is examined, the $l_{occ}$ of its children are removed, merged and the new $l_{occ}$ is attached to the father. Moreover, two nodes, one of which is not the ancestor of the other, do not share any occurrence position. As the leaves contains all $n$ positions of the sequence, at any step all $l_{occ}$ stored contain at most all the positions of the leaves, i.e. $n$. Since each time a factor is selected, its position is deleted and then put into $\mathcal{F}$, the union of those lists has a decreasing number of elements ($n$ at the beginning), and $\mathcal{F}$ is its complement to the set of all positions (formally: $Card(\mathcal{F}) + \sum Card\ (\mathrm{l_{occ}}) = \mathrm{n}$.)
4. Factors in $\mathcal{F}$ are longer than $\log(n)$ and are not overlapping each other. Thus $\mathcal{F}$ contains a maximum of $O(n/log(n))$ elements.

We thus have an $O(n)$ space complexity. *Compute_ST* takes linear time [8] and to sort $L_R$ is done in $O(n \log(n))$. Inside the main loop, *compute_list_occurrences* merges $|\mathcal{A}|$ lists of occurrences and thus takes at most $O(n)$ steps. The while loop goes through $l_{occ}$ and $\mathcal{F}$ which are at most $n$ long, and the computation of the overlap is done in constant time. So the main loop requires at most $O(n)$ steps and is performed at most $n$ (number of repeats in $L_r$) times. We end up with an $O(n^2)$ time complexity. As the encoding is linear in $n$, these are also the overall complexities. □

# 4 Applications and conclusions

We explain first why the combination of repeats reported by *Search_Repeats* on a compressed sequence is significant. This sheds light on how a compression test should be interpreted. In a second part, we report on two examples of repeats from the application of *Search_Repeats* to the study of *Haemophilus influenzae* genome.

## 4.1 Compression and non-randomness.

Random sequences can enclose repeats, even arbitrarily long ones. But on average, the size of their longest repeat is in $O(\log(n))$ [5]. The code for a repeat is also in $O(\log(n))$. Therefore, on average a repeat from a random sequence does not yield a positive gain if it is encoded. Shortly, we can say: “random is incompressible”. It is known that at most one sequence out of one million random sequences of length $n$ is compressible of 20 bits or more; a compression of more than 20 bits is therefore significant. If one inputs a genetic sequence in *Search_Repeats* and finds that it is compressed, one can draw the following conclusion: the combination of repeats detected by *Search_Repeats* would have probably not been found in a random sequence. And the higher the compression gain, the more unprobable it would be. This is the reason why compressing is a test of non-randomness (those ideas are developped in [14, 17].)

For the sake of comparison, we give an example of an imaginary DNA sequence and its compression gain. The unique repeat (12 bp) is very long compared to the sequence length (40 bp), but the compression rate is nevertheless very low: 5% (especially compared to usual compression of normal files where it often reaches 50%.) It denotes first that the compression rate value can be misleading and second what is often the case with genetic sequences: they do not contain more than 25% of repeats like in our example, and are therefore not very compressible [18].

$$s = \text{AGTACATATAGTCGCATACGCTGCAATAGTCGCATACATG}$$

**Segment code:** "1,(8,12,6),AGTACATATAGTCGCATACGCTGCAATG"

**Compression gain/rate:** 4 bits (76 bits versus 80 bits), nearly 5% of compression rate

### 4.2 Application to *Haemophilus influenzae.*

The repetitive structure of DNA is particularly interesting to study for long sequences, like complete chromosomes or whole genomes. We ran *Search_Repeats* on the same Sun Ultrasparc workstation than for our BLASTN tests with the genome of *Haemophilus influenzae* as input (1.8 Mbp). In comparison, it took 3 seconds of CPU time instead of more than 8 hours with BLASTN.

With a minimal size of 19 bp for the repeats, the algorithm replaces 33365 bp belonging to repeats by 10143 bits of code (for the list of pointers only), which results in a compression gain of 56577 bits and a compression rate of only 1.55% of reduction. 210 exact repeats were output ranging in length from 5563 bp to 19 bp. Most of them belong to a *cluster*. A cluster is in reality an approximate repeat which encloses several exact sub-repeats, and those occur together at the same locations in the same order. The table below gives an example of a 4 repeats cluster. For instance, the source occurrences of the first two repeats are only separated by 1 bp ($= 574300 - 574298 - 1$), as well as their reference occurrences; the difference of those offsets is the phase (with value 0.) It means that they form an approximate repeat with at most 1 mutation between them but no indels.

| | Source | | Target | | |
|---|---|---|---|---|---|
| length | begin | end | begin | end | phase |
| 65 | 574234 | **574298** | 574399 | **574463** | – |
| 69 | **574300** | 574368 | **574465** | 574533 | **0** |
| 38 | 574373 | 574410 | 574538 | 574575 | 0 |
| 101 | 574412 | 574512 | 574577 | 574677 | 0 |

*Search_Repeats* allows to see insertion/deletion of long segments between two subsequent repeats inside a cluster. This is the case when the offset between source occurrences is small and the offset between target occurrences is as large as the inserted segments (or vice-versa.) For instance, a cluster in which all except 2 of the 48 repeats are in perfect phase (0 indels) and the two others reveal each a large insertion of 899 and 4195 bp respectively (in between the source occurrences.) This approximate repeat of 3.2 kbp is also reported in [12] with the same insertions (the algorithm used in the one of Leung and al. [13].)

Another example is an approximate repeat of 5563 bp which occurs 3 times as a direct repeat and 3 times as an inverted repeat (data not shown.) One inverted and two direct occurrences contain a deletion of a 245 bp. It implies that several duplication events happened before and after the deletion. This repeats contains genes for various RNAs and are not reported in [12]. This example raises many questions about the date and the mechanism of such a duplication.

**Conclusion**

We presented an algorithm which detects repeats that corresponds to duplication events, without restriction on the length nor on the spacing of the occurrences. It allows to assert the presence of large insertion/deletion inside those repeats, which cannot be achieved with algorithms looking for local similarity. The speed of *Search_Repeats* makes it possible to analyze complete genomes or chromosomes in very little time.

**Acknowledgements:** We thank gratefully Herrn Bornberg, Heber, Müller, Spang and Vingron for commenting the manuscript. E. Rivals is partly supported by grant 01 KW 9601 of the Human Genome Project from the German Ministry of Research, an MRES allocation from the French Ministry of Research. This work was also supported the G.D.R. CNRS 1029 and by the "Groupement de Recherches et d'Etudes des Génomes".

## References


[1] Pankaj Agarwal and David J. States. The repeat pattern toolkit (rpt): Analyzing the structure and evolution of the c. elegans genome. In *Proc. of the Second International Conference on Intelligent Systems for Molecular Biology*, pages 2–9, 1994.

[2] Stephen F. Altschul, Warren Gish, Webb Miller, Eugene W. Myers, and David J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215:403–410, 1990.

[3] A. Apostolico and A.S. Fraenkel. Robust transmission of unbounded strings using Fibonacci representations. *IEEE Trans. Inform. Theory*, 33(2):238–245, 1987.

[4] Alberto Apostolico. The myriad virtues of subword tree. In A. Apostolico and Z. Galil, editors, *Combinatorial Algorithms on Word*. Springer-Verlag, 1985.

[5] Richard Arratia, Louis Gordon, and Michael Waterman. An extreme value theory for sequence matching. *The Annals of Statistics*, 14(3):971–993, 1986.

[6] Alvis Brazma, Inge Jonassen, Ingvar Eidhammer, and David Gilbert. Approaches to the automatic discovery of patterns in biosequences. Technical Report 113, Dept. of Informatics, Univ. of Bergen, Norway, 1995.

[7] E. Mac Creight. A space-economical suffix tree construction algorithm. *Journal of the Association of Computing Machinery*, 23(2):262–272, April 1976.

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

[9] Olivier Delgrange. *Un algorithme rapide pour une compression modulaire optimale. Application à l'analyse de séquences génétiques.* PhD thesis, Université des Sciences et Technologies de Lille, 1997.

[10] R. D. Fleischmann, M. D. Adams, O. White, R. A. Clayton, E. F. Kirkness, A. R. Kerlavage, C. J. Bult, J. F. Tomb, B. A. Dougherty, and J. M. Merrick. Whole-genome random sequencing and assembly of Haemophilus influenzae Rd. *Science*, 269:496–512, 1995.

[11] Stéphane Grumbach and Fariza Tahi. A New Challenge for Compression Algorithms: Genetic Sequences. *Journal of Information Processing and Management*, 30(6):875–866, 1994.

[12] Samuel Karlin. Assessing Inhomogeneities in Bacterial Long Genomic Sequences. In Michael Waterman, editor, *Proc. of the First Annual International Conference on Computational Molecular Biology*, pages 164–171. ACM Press, 20-23 January 1997.

[13] M-Y. Leung, B.E. Blaisdell, C. Burge, and S. Karlin. An Efficient Algorithm for Identifying Matches with Errors in Multiple Long Molecular Sequences. *Journal of Molecular Biology*, 221:1367–1378, 1991.

[14] Ming Li and Paul M.B. Vitanyi. *Introduction to Kolmogorov Complexity and Its Applications*. Springer-Verlag, 2nd edition edition, 1997.

[15] Aleksandar Milosavljeviċ and Jerzy Jurka. Discovering Simple DNA Sequences by the Algorithmic Significance Method. *CABIOS*, 9(4):407–411, 1993.

[16] William R. Pearson and David J. Lipman. Improved tools for biological sequence comparison. *PNAS*, 85:2444–2448, 1988.

[17] É. Rivals, M. Dauchet, J-P. Delahaye, and O. Delgrange. Compression and genetic sequences analysis. *Biochimie*, 78(4):315–322, 1996.

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

[19] Temple F. Smith and Michael S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147:195–197, 1981.

[20] J.A. Storer. *Data Compression: Methods and Theory.* Computer Sciences Press, 1988.

[21] R.D. Wells and R.R. Sinden. *Genome Rearrangement and Stability*, volume 7 of *Genome Analysis*, chapter Defined Ordered Sequence DNA, DNA Structure, and DNA-directed Mutation. Cold Spring Harbor Laboratory Press, 1993.


### Appendix A: the suffix tree data structure.

The suffix-tree (ST) is a data structure which stores all factors of a text. It is built in $O(n)$ time and requires $O(n)$ space. An example is shown in figure 7. For detailed presentations see [4, 8] among many articles. The ST is a rooted tree whose nodes represent the different substrings of the text. It requires that a character not in $\mathcal{A}$ is appended to the text in such a way that any suffix cannot be the prefix of another one, i.e. all suffixes become unique substrings. Each leaf is mapped to a suffix (and reversely) and labeled by its beginning position. Each path from the root to a leaf represents a suffix by the concatenation of the edges labels on this path. Therefore, paths from the root to successive internal nodes represent longer and longer prefixes of this suffix. Two other properties state that i) the first letter of sisters-edges must be different and ii) an internal node has at least two children. This induces that the lowest ancestor of two leaves represent the longest common prefix between the two suffixes. Thus, an internal node is mapped to a repeated substring, but not in a one-to-one manner: only a *maximal*[8] repeated substring is associated to an internal node. In figure 7, it can be seen that "caga" is mapped to a node while "cag" is not. The latter always appears enclosed in an occurrence of the former. At last, the set of positions of occurrences of a substring is the set of labels of all leaves in subtree.

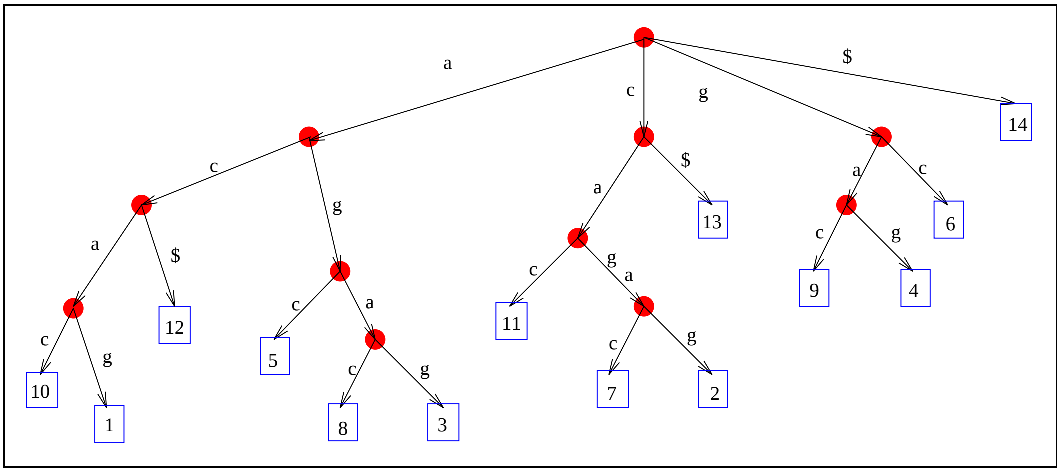


Figure 7: The Suffix Tree for the text "acagagcagacac$" where $ is the end-marker. "caga" is a maximal repeated substring while "cag" is not. In fact, only the first letter of an edge label is really stored, and in a node only a couple of integers giving the position of occurrence and the substring length are recorded. Occurrence positions of "ac" are $1, 10, 12$ and $3, 5, 8$ for "ag".

[8]More precisely, it is a right maximal substring in the ST.