



ELSEVIER

Information Processing Letters 83 (2002) 1–6

Information  
Processing  
Letters

www.elsevier.com/locate/ipl

# Compror: On-line lossless data compression with a factor oracle<sup>☆</sup>

Arnaud Lefebvre<sup>a,\*</sup>, Thierry Lecroq<sup>b</sup>

<sup>a</sup> UMR CNRS 6037–ABISS, Faculté des Sciences et Techniques, Université de Rouen, 76821 Mont-Saint-Aignan Cedex, France

<sup>b</sup> LIFAR–ABISS, Faculté des Sciences et Techniques, Université de Rouen, 76821 Mont-Saint-Aignan Cedex, France

Received 21 June 2001; received in revised form 22 October 2001

Communicated by L. Boasson

---

## Abstract

We present in this article a linear time and space data compression method. This method, based on a factor oracle and the computation of the length of repeated suffixes, is easy to implement, fast and gives good compression ratios. © 2001 Elsevier Science B.V. All rights reserved.

*Keywords:* Combinatorics on word; String algorithms; Repetitions; Factor oracle; Suffix link; Data compression; Algorithms; Combinatorial problems; Formal languages

---

## 1. Introduction

Compression serves both to save storage place and to save transmission time. The interest in data compression techniques remains important even if mass storage systems improve regularly because the amount of data grows accordingly. Moreover, a consequence of the extension of computer networks is that the quantity of data they exchange grows exponentially. So, it is often necessary to reduce the size of files to reduce proportionally their transmission times.

Different strategies have already been developed such as statistical methods [5,8] or methods based on dictionaries [10,11]. Zipstein [9] designed a method

based on factor automata to find the repetitions. More recently, methods based on combinatorial properties were given: one based on the Burrows–Wheeler transform [3] and another one based on antictionaries [2].

In [6] we describe an on-line method, linear in time and space, to compute, for each prefix of a word  $p$ , the length and an occurrence of one of its repeated suffixes. This method, based on a factor oracle [1], gives very good results in the search of repetitions in genomic sequences. Since this method gives, for each prefix of a word  $p$ , the length and an occurrence of one of its repeated suffixes, we get a natural on-line data compression scheme.

In Section 2 we describe the factor oracle and the computation, for each prefix of a word, of the length and a position of a repeated suffix. Then, in Section 3, we explain the principles of *compror*, the compression method we developed. Section 4 contains the encoding details. Experimental results are shown in Section 5. Finally, Section 6 presents our conclusions.

---

<sup>☆</sup> This work was partially supported by a NATO grant PST.CLG.977017.

\* Corresponding author.

*E-mail addresses:* Arnaud.Lefebvre@univ-rouen.fr

(A. Lefebvre), Thierry.Lecroq@univ-rouen.fr (T. Lecroq).

*URLs:* <http://al.jalix.org> (A. Lefebvre),

<http://www-igm.univ-mlv.fr/~lecroq> (T. Lecroq).

## 2. Computing the length and the position of repeated suffixes

The method described in [6], based on a factor oracle (see [1]), gives, in linear time and space, the length and an occurrence of a repeated suffix of each prefix of a word  $p$  (see Fig. 2). In this section, after a few basic definitions, we briefly present this result.

### 2.1. Definitions

Let  $p = p[1..m]$  be a word of length  $|p| = m$  over an alphabet  $\Sigma$ . Let  $\varepsilon$  be the empty word ( $|\varepsilon| = 0$ ). A word  $w \in \Sigma^*$  is a *factor* of  $p$  if and only if  $p$  can be written  $p = uvw$  with  $u, v \in \Sigma^*$ . A word  $u \in \Sigma^*$  (respectively  $v \in \Sigma^*$ ) is a *prefix* (respectively *suffix*) of  $p$  if and only if  $p$  can be written  $p = uv$  with  $u, v \in \Sigma^*$ . An *occurrence* of a factor  $w$  of  $p$  is denoted by the position  $i \in [1..m]$  of its ending letter. A *repeated factor* of a word  $p$  is a factor of  $p$  which has at least two distinct occurrences in  $p$ .

### 2.2. The factor oracle

The factor oracle of a word  $p$  of length  $m$ , denoted by  $Oracle(p)$ , is a deterministic finite automaton  $(Q, q_0, F, \delta)$  where  $Q = \{0, 1, \dots, m\}$  is the set of states,  $q_0 = 0$  is the starting state,  $F = Q$  is the set of terminal states and  $\delta$  is the transition function. The factor oracle of a word  $p$  of length  $m$  has the following properties: it has exactly  $m + 1$  states; it has within  $m$  and  $2m - 1$  transitions; it recognizes at least all the factors of  $p$ . The exact characterization of the language recognized by the factor oracle is still an open question.

There is a bijection between the states of the oracle and the  $m + 1$  prefixes of  $p$  (including the empty one). Each transition leading to state  $i$  is labeled by  $p[i]$ . We distinguish two kinds of transitions: transitions from state  $i$  to state  $i + 1$  are called *internal transitions* and

transitions from state  $i$  to state  $j$  such that  $j - i > 1$  are called *external transitions*, with  $0 \leq i < j \leq m$ . There are exactly  $m$  internal transitions. Thus, to store the oracle, one needs to store only the word  $p$  and at most  $m - 1$  external transitions without their label. All the other informations can be deduced from the word  $p$ . This representation is completely independent from the underlying alphabet and is very economical. This structure is linear in space, and its construction is linear in time (see proof in [1]). Fig. 1 shows the oracle of the word  $aabbabbabbab$ .

We can use the factor oracle of a word  $p$  to compute the length of long repeated suffixes for each prefix of  $p$ . Let us introduce some definitions.

**Definition 1.** We denote by  $LRS(i)$  the longest repeated suffix of  $p[1..i]$ :

$$LRS(i) = \max\{v \mid v \text{ is a suffix of } p[1..i] \text{ and } v \text{ is a factor of } p[1..i-1]\}.$$

**Definition 2** [1].  $S[i]$ , the suffix link of a state  $i$  of  $Oracle(p)$ , is equal to the state in which the longest repeated suffix of  $p[1..i]$  is recognized:

$$S[i] = \delta(0, LRS(p[1..i])).$$

The state  $S[i]$  is equal to an occurrence of a repeated suffix of  $p[1..i]$ .

**Definition 3** [1]. For  $0 < i \leq m$ ,  $SP(i) = (k_0 = i, \dots, k_t = 0)$  is the suffix path of state  $i$  in  $Oracle(p)$ , such that for all  $r$ ,  $1 \leq r \leq t$ ,  $k_r = S[k_{r-1}]$ .

We can now define the array  $lrs$  where  $lrs[i]$ ,  $0 \leq i \leq m$ , is a good approximation of  $|LRS(i)|$ .

The reader can refer to [6] for the details of the computation of the  $lrs$  values. We proved in [6] that for each state  $i$  of  $Oracle(p[1..|p|])$ ,  $lrs[i]$  is equal to the length of a repeated suffix of  $p[1..i]$  such that

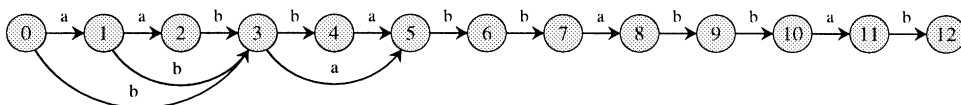


Fig. 1. Factor oracle of the word  $aabbabbabbab$ . All the factors of  $aabbabbabbab$  are recognized from state 0. The word  $aba$  is recognized though it is not a factor of  $aabbabbabbab$ . This factor oracle can be represented by  $aabbabbabbab$ ,  $(0, 3)$ ,  $(1, 3)$  and  $(3, 5)$ .

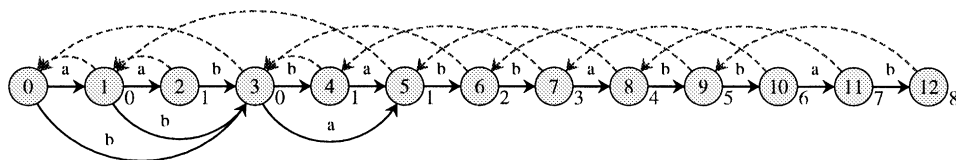


Fig. 2. Example of computation of *lrs*. The dashed arrows represent the suffix links and the plain arrows the transitions. The values written near the states are the *lrs* values.

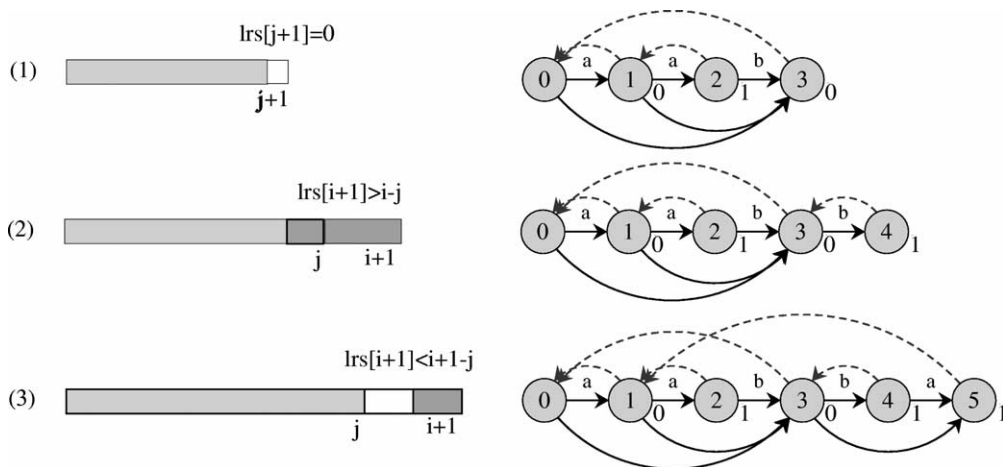


Fig. 3. (1) Factor *aa* has already been encoded then a new letter *b* is encoded. (2) Factor *aab* has already been encoded. Factor *b* has another occurrence ending in position 3: *b* will not be encoded yet. (3) It is the first occurrence of factor *ba* because  $lrs[5] = 1$  (not 2), thus *b* will be encoded.

one of its occurrences is equal to  $S[i]$ . Furthermore, experimental results on biological data show that *lrs* values constitute a very good approximation of *LRS* values. An example is given Fig. 2. The fact that we are able to find an occurrence and the length of a repeated suffix for each prefix of a word, leads to develop an on-line lossless text compression method.

### 3. The compression model

All the first occurrences of the letters of the alphabet will be encoded as single letters. All the repeated factors will be encoded as pairs (*length, position*). The encoding will be performed simultaneously with the construction of the factor oracle.

Assume that  $Oracle(p[1..j])$  has already been built and thus that the prefix  $p[1..j]$  of length *j* of the word *p* has already been encoded. To encode the suffix  $p[j+1..m]$ , we then need to find the smallest position

$i + 1$  strictly greater than *j* such that  $lrs[i + 1] < i + 1 - j$  (see Fig. 3). If  $i + 1 = j + 1$  (or equivalently  $lrs[j + 1] = 0$ ) then it means that the letter  $p[j + 1]$  never occurred in  $p[1..j]$  and it will be encoded as a single letter. Otherwise we will represent  $p[j + 1..i]$  as the pair  $(i - j, S[i] - i + j + 1)$  since  $p[j + 1..i] = p[S[i] - i + j + 1..S[i]]$ . At that time the prefix  $p[1..i]$  has been encoded, it remains to encode by the same process the suffix  $p[i + 1..m]$ .

The word *aabbabbabbab* which oracle and *lrs* values are given Fig. 2 will be encoded by  $a(1, 1)b(1, 3)(8, 2)$ .

The decoding process is straightforward. Given  $a(1, 1)b(1, 3)(8, 2)$  it is obvious to retrieve the word *aabbabbabbab*.

### 4. Encoding details

As described in the previous section, two types of objects are encoded: a single letter *a* is encoded by a

pair  $(0, \text{ASCII}(a))$ , and a repeated suffix is encoded by a pair  $(\text{length}, \text{position})$ .

#### 4.1. General scheme

The general scheme of our compression method is given Fig. 4. It is a Lempel–Zip like scheme. Consider that  $p[1..j]$  has already been encoded. Let us build the factor oracle of  $p$  from state  $j$  until we find a position  $i + 1 > j$  where  $\text{lrs}[i + 1] < i + 1 - j$ . In this case, position  $i + 1$  is sent to the encoder. Two cases are then considered:

- $i + 1 = j + 1$  (thus  $\text{lrs}[i + 1] = 0$ ): the letter encoder sends a pair  $(0, \text{letter})$  to the pair encoder;
- $i > j$  (thus  $\text{lrs}[i] > 0$ ): a pair  $(\text{length}, \text{position})$  is sent to the pair encoder.

In the second case, a natural test is done: if the length of the code of the pair is longer than encoding  $\text{length}$  times a letter, each letter of the repeated suffix is sent to the letter encoder.

In the next subsections, the letter encoder, the way we find the best pair to encode and the pair encoder are described.

#### 4.2. Encoding a letter

As the algorithm does not depend on the alphabet, all the letters of the ASCII code can be encoded. Thus, a letter  $a$  is encoded by the pair  $(0, \text{ASCII}(a))$  where the 0 indicates that the next value is not a position but the ASCII code of a letter.

#### 4.3. Encoding a repeated suffix

Consider that we are going to encode the repeated suffix of  $p[1..i]$ , and that the prefix  $p[1..j]$  has already been encoded. In all cases, we encode the length  $i - j$ . We consider that a position is better than another if the length of its code is shorter than the length of the code of the other one. Naturally, the smaller a position is, the shorter its code is. Now, the question is: is  $S[i] - i + j + 1$  the best position to encode? There are two cases:

- if  $\text{lrs}[i] = i - j$ , the repeated suffix which is encoded does not overlap the prefix already encoded. Since along  $SP(i)$ , from  $i$  to 0, the values of  $\text{lrs}$  are strictly decreasing (see [6]),  $S[i] - i + j + 1$  is the best position we can encode;
- if  $\text{lrs}[i] > i - j$ , the repeated suffix which is encoded overlaps the prefix already encoded (see Fig. 5). In this case, we can follow  $SP(i)$  from  $i$  to 0 until we find a state  $k$  such that the following two conditions hold:

$$\text{lrs}[k] \geq i - j \quad \text{and} \quad \text{lrs}[S[k]] < i - j.$$

Thus, the position we encode is  $S[k] - i + j + 1 < S[i] - i + j + 1$ , and consequently, the length of its code could be smaller.

#### 4.4. Encoding the pairs

The pairs  $(\text{length}, \text{position})$  are encoded using order 2 and 3 Fibonacci codes described in [4]. This series can be seen as a base. If no consecutive numbers

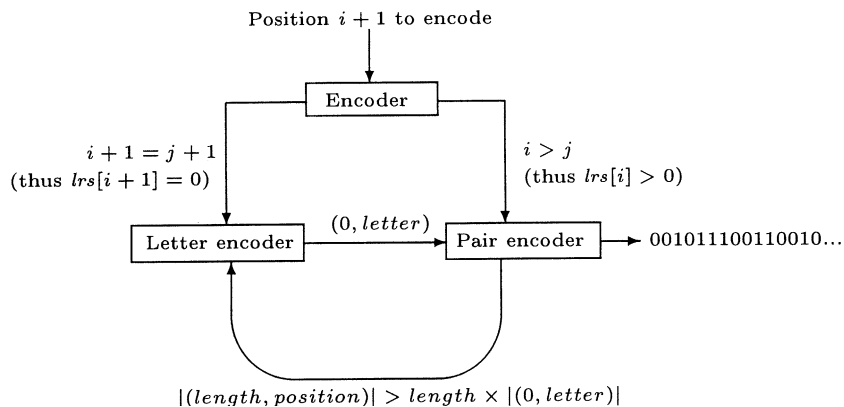


Fig. 4. General scheme of compror.

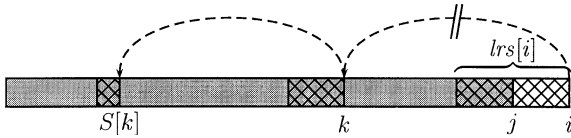


Fig. 5. Looking for the best position to encode: dashed arrows are suffix links, crosshatched parts show the  $lrs$  values of the indicated positions,  $k \in SP(i)$  and the position we encode is  $S[k] - i + j + 1$ .

of this series are used, the decomposition of any strictly positive integer is unique. The decomposition is transformed in a self-delimiting code by appending a 1 to the rightmost 1. The 3 order Fibonacci code is obtained with the same technics using the order 3 series  $F_0^3 = 0, F_1^3 = 1, F_2^3 = 2$  and  $F_i^3 = F_{i-1}^3 + F_{i-2}^3 + F_{i-3}^3$  for  $i \geq 3$ . As we can see in Fig. 6, the lengths of order 3 Fibonacci codes (sometimes called tribonacci codes) become shorter than order 2 Fibonacci codes for values greater than 100. In most cases, the lengths to encode are smaller than 100 while the encoded positions are greater than 100. That is why lengths are encoded with order 2 Fibonacci codes and positions with order 3 Fibonacci codes.

#### 4.5. Improvement

A simple artfulness improves the way of encoding Fibonacci codes.

**Proposition 1.** *The length of the order 2 Fibonacci code of an integer  $n \geq 1$  is strictly greater than the length of the order 2 Fibonacci code of the quotient of the integer division of  $n$  by 2.*

**Proof.** The lengths of order 2 Fibonacci codes of two integers are different if and only if there exists a Fibonacci number between them. Let us prove that  $\forall n > 0, \exists i > 0, n \leq F(i) < 2n$  (or  $n < F(i) \leq 2n$ ). By contradiction let us assume that  $\exists n > 0, \exists i > 0$  such as  $F(i) \leq n < 2n < F(i + 1)$ . We can write  $n = F(i) + k$  where  $0 \leq k < F(i + 1)$ . Thus we obtain  $2n = 2F(i) + 2k$ . Consequently we have  $2F(i) + 2k < F(i) + F(i - 1)$  and finally,  $F(i) + 2k < F(i - 1)$  which is impossible.  $\square$

Thus, we encode the quotient of the integer division of  $n$  by 2 plus its remainder (0 or 1), instead of  $n$  itself: in most cases, we gain one bit per encoded integer.

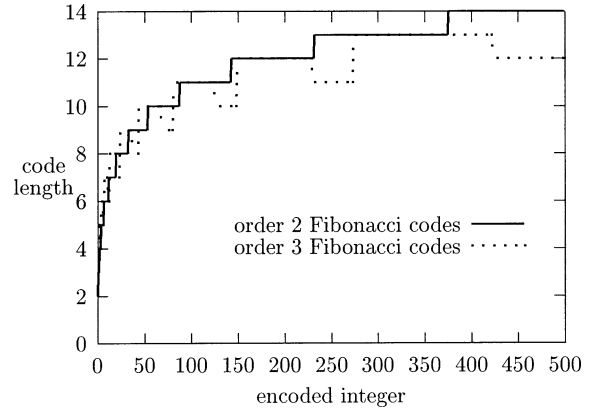


Fig. 6. Curves representing the lengths of order 2 Fibonacci and order 3 Fibonacci codes for values between 1 and 500: order 3 Fibonacci codes become shorter than order 2 Fibonacci codes for values greater than 100.

We can apply the same technics to order 3 Fibonacci codes.

## 5. Experimental results

We compared our method to two well-known data compression methods, `gzip` [10,11] and `bzip2` [3]. The experiments were done on a PC running Linux with two 500 MHz processors and 1 GB RAM. The results, given in Fig. 7, show that `compror` is a fast method which gives good compression ratios. The best results are obtained on files containing  $\LaTeX$  sources, postscript files and encapsulated postscript figures. In these cases, the problems for `gzip` may be the sizes of the window and the dictionary and also collisions with the hashing function. For `bzip2` it comes from the fact that it uses a maximal size of block of 900 kb.

## 6. Conclusion and open questions

We present in this article a compression method using the factor oracle which is linear in time and space. The main interests of this method is that it is easy to understand and to implement, it also proves to be very fast in practice. The results we obtain may be improved, in particular in the case of genomic sequences (built on a four or twenty letter alphabet) (see [7]). Also, it should be possible to improve

file	size	gzip			compror			bzip2		
		bpc	ct	dt	bpc	ct	dt	bpc	ct	dt
ChrIV	17,550,082	2.17	230	<b>2</b>	2.9	<b>37</b>	9.65	<b>2.13</b>	<b>37</b>	9.65
ChrII	19,647,091	2.18	261	<b>2</b>	2.9	42	10.9	<b>2.14</b>	<b>40</b>	12
bib	111,261	2.5	<b>0.14</b>	<b>0.01</b>	3.1	0.21	0.06	<b>1.97</b>	0.17	0.04
book1	768,771	3.25	<b>1.18</b>	<b>0.08</b>	3.8	2.00	0.5	<b>2.42</b>	1.47	0.47
book2	610,856	2.7	<b>0.68</b>	<b>0.06</b>	3.3	1.30	0.35	<b>2.06</b>	1.1	0.33
progc	39,611	2.67	<b>0.04</b>	<b>0.01</b>	3.8	0.09	0.02	<b>2.53</b>	0.07	0.02
trans	93,695	1.61	<b>0.08</b>	<b>0.01</b>	2.16	0.13	0.04	<b>1.52</b>	0.14	0.03
thesis	8,284,160	1.74	<b>7.47</b>	<b>0.65</b>	<b>0.94</b>	7.65	2.37	1.3	18.6	3.44
article1	2,764,800	1.65	<b>5.14</b>	<b>2.5</b>	<b>1.2</b>	37.5	<b>2.5</b>	1.65	<b>5.14</b>	<b>2.5</b>
article2	8,181,760	2.2	24.6	<b>7.1</b>	<b>1.25</b>	51.5	8.2	2.04	<b>8.8</b>	8.6
alpha	530,000	0.025	<b>0.08</b>	0.57	<b>0.002</b>	0.3	<b>0.55</b>	0.003	4.45	0.6

Fig. 7. Data compression results: our method has been compared to `gzip` and `bzip2`. Files `ChrII` and `ChrIV` contain the DNA sequences of chromosomes II and IV of *Arabidopsis thaliana*. Files `bib`, `book1`, `book2`, `progc` and `trans` are taken from the Calgary Corpus and are respectively a bibliography, two English texts, a C program and a transcript of a terminal session. Files `thesis`, `article1` and `article2` are archive files containing L<sup>A</sup>T<sub>E</sub>X sources, postscript files and encapsulated postscript figures. File `alpha` consists in 10,000 lines containing `abcd...zABCD...Z`. Sizes of each file are given in bytes in the second column. For each method we give the compression ratio (bpc) in bits per input character, encoding time (ct) and decoding time (dt) are then given in seconds.

the different encoders. The factor oracle should be used to develop an efficient off-line compression scheme. An implementation of `compror` is available at <http://al.jalix.org>.

## Acknowledgements

We are grateful to Marc Zipstein whose comments and remarks greatly improved the quality of this article.

## References

- [1] C. Allauzen, M. Crochemore, M. Raffinot, Factor oracle: A new structure for pattern matching, in: J. Pavelka, G. Tel, M. Bartosek (Eds.), SOFSEM'99, Theory and Practice of Informatics, Milovy, Czech Republic, Lecture Notes in Comput. Sci., Vol. 1725, Springer, Berlin, 1999, pp. 291–306.
- [2] M. Crochemore, F. Mignosi, A. Restivo, S. Salemi, Text compression using antidictionaries, in: Proceedings of the 26th International Colloquium on Automata, Languages and Programming, Lecture Notes in Comput. Sci., Vol. 1644, Springer, Berlin, 1999, pp. 261–270.
- [3] P. Fenwick, The Burrows–Wheeler transform for block sorting text compression—Principles and improvements, *Comput. J.* 39 (9) (1996) 731–740.
- [4] A.S. Fraenkel, S.T. Klein, Robust universal complete codes for transmission and compression, *Discrete Appl. Math.* 64 (1996) 31–55.
- [5] D.-A. Huffman, A method for the construction of minimum redundancy codes, *Proc. I.R.E.* 40 (9) (1952) 1098–1101.
- [6] A. Lefebvre, T. Lecroq, Computing repeated factors with a factor oracle, in: L. Brankovic, J. Ryan (Eds.), Proceedings of the 11th Australasian Workshop On Combinatorial Algorithms, Hunter Valley, Australia, 2000, pp. 145–158.
- [7] C.G. Nevill-Manning, I.H. Witten, Protein is incompressible, in: J.A. Storer, M. Cohn (Eds.), Proceedings of the Data Compression Conference, IEEE Press, Los Alamitos, CA, 1999, pp. 257–266.
- [8] I.H. Witten, R. Neal, J. Cleary, Arithmetic coding for data compression, *Comm. ACM* 30 (6) (1987) 520–540.
- [9] M. Zipstein, Data compression with factor automata, *Theoret. Comput. Sci.* 92 (1) (1992) 213–221.
- [10] J. Ziv, A. Lempel, A universal algorithm for sequential data compression, *IEEE Trans. Inform. Theory* 23 (1977) 337–343.
- [11] J. Ziv, A. Lempel, Compression of individual sequence via variable length coding, *IEEE Trans. Inform. Theory* 24 (1978) 530–536.