Window subsequence problems for compressed texts*

Patrick Cégielski[1], Irène Guessarian[2], Yury Lifshits[3], Yuri Matiyasevich[3]

[1] LACL, UMR-FRE 2673, Université Paris 12, Route forestière Hurtault, F-77300 Fontainebleau, France,
cegielski@univ-paris12.fr

[2] LIAFA, UMR 7089 and Université Paris 6, 2 Place Jussieu, 75254 Paris Cedex 5, France; send correspondence to ig@liafa.jussieu.fr

[3] Steklov Institute of Mathematics, Fontanka 27, St. Petersburg, Russia.
yura@logic.pdmi.ras.ru, yumat@pdmi.ras.ru

Abstract. Given two strings (a text t of length n and a pattern p) and a natural number w, *window subsequence problems* consist in deciding whether p occurs as a subsequence of t and/or finding the number of size (at most) w windows of text t which contain pattern p as a subsequence, *i.e.* the letters of pattern p occur in the text window, in the same order as in p, but not necessarily consecutively (they may be interleaved with other letters). We are searching for subsequences in a text which is compressed using Lempel-Ziv-like compression algorithms, without decompressing the text, and we would like our algorithms to be almost optimal, in the sense that they run in time $O(m)$ where m is the size of the compressed text. The pattern is uncompressed (because the compression algorithms are evolutive: various occurrences of a same pattern look different in the text).

1 Introduction

We are concerned with searching information in a compressed text *without decompressing* the text. We will search to decide whether a pattern occurs as a *subsequence* of a text: pattern $p = p_1 \dots p_k$ is said to be a subsequence of text t if $p_1, \dots, p_k$ occur in t, in the same order as in p, but not necessarily consecutively (they may be interleaved with other letters). It is also demanded that the subsequences consisting of p be contained in text windows of (at most) a fixed size w. Pattern matching in compressed texts has already been studied in *e.g.* [R99,GKPR96]. Subsequence matching within windows of size w at most is a more difficult problem, which emerged due to its applications in knowledge discovery and datamining [M02], and as a first step for solving problems in molecular biology. One quite important use of subsequence matching consists

* Support by grants INTAS–04-77-7173 and NSh–2203-2003-1 is gratefully acknowledged.

in recognizing frequent patterns in large sequences of data. Knowledge of frequent patterns is then used to determine association rules in databases and to predict the behavior of large data. Consider for instance a text t consisting of a university WWW-server logfile containing requests to see WWW pages, and suppose we want to see how often, within a time window of at most 10 units of time, the sequence of events $e_1e_2e_3e_4$ has occurred, where: e_1 = 'Computer Science Department homepage', e_2 = 'Graduate Course Descriptions', e_3 = 'CS586 homepage', e_4 = 'homework'. This will be achieved by counting the number of 10-windows of t containing $p = e_1e_2e_3e_4$ as a subsequence.

Most efficient compression algorithms are evolutive, in the sense that the text represented by each compression symbol is determined dynamically, hence the encoding of a subword is different for different occurrences of this subword in the text. It is thus less useful for the search to encode the pattern. Moreover, pattern sizes are usually smaller than text sizes by several orders of magnitude. We will thus search for a plain (not encoded) pattern in an encoded (compressed) text.

We address several *window subsequence* problems in three models of compression: Lempel-Ziv (in short LZ [LZ77]), Lempel-Ziv-Welch (in short LZW [LZ78,W84]), and straight-line-programs (in short SLP [R03]). We show that, for all three models, as soon as there is a significant (say quadratic) difference in size between the compressed text and the original text, searching directly in the compressed text is more efficient than the naive decompress-then-search approach.

The paper is organised as follows: in Section 2, we recall the compression models, in Section 3 we define five window subsequence problems, in Section 4 we describe auxiliary data structures, and show how to compute them, yielding algorithms for the window subsequence problems.

Related results

Different versions of pattern-matching and subsequence problems have been considered. Subsequence problems are different from pattern-matching problems in two respects: 1. the letters of the pattern need not be consecutive in the text, and 2. the size of the text window where the pattern occurs is bounded. Some related problems are as follows.

It was shown in [GKPR96] that the pattern matching problem can be solved in polynomial time, even if *both* text and pattern are given in LZ compressed form.

It was shown in [L05,LL05] that problem 1 below (Section 3), is both NP and co-NP-hard if *both* text and pattern are given in LZ compressed form.

It was shown in [ABF95] that finding the first occurrence of the pattern (pattern matching with compressed text and uncompressed pattern) in an LZW-style compressed text can be done in time $O(m + k^2)$ or $O(m \log k + k)$.

Next, we should mention [BKLPR02], where *Compressed Pattern Matching* problems were extended to the two-dimensional case: it was shown that complexity increases in this setting. *Compressed Pattern Matching* is NP-complete while *Fully Compressed Pattern Matching* is Σ_2^P-complete.

Recently, applications for algorithms on compressed texts in analysis of message sequence charts were found, see [GM02].

Besides pattern matching the membership of a compressed text in a formal language was studied. In [GKPR96], the authors presented a polynomial algorithm for deciding membership in a regular language. Recently, [MS04] showed that this problem is P-complete. On the other hand, it was shown in [Loh04] that deciding membership in a context-free language is PSPACE-complete.

2 Compression algorithms

2.1 Notations

An *alphabet* is a finite non-empty set $A = \{a_1 \ldots, a_i\}$. We will also use an extra letter a_0. A *word* t on A is a sequence $t[1]t[2]\cdots t[n]$ of letters from A (also denoted by $t_1t_2\cdots t_n$ and called the *text*). The number n is called the *length* of t and will be denoted by $|t|$. The only length zero word is the *empty word*, denoted by ε. Given integers $k \leq n$, $i < j \leq n$ and t a length n word, let $t[k]$ (resp. $t[-k]$, $t[i..j]$) denote the kth leftmost letter of t (resp. the kth rightmost letter, the subword $t[i]t[i+1]\cdots t[j]$ of t).

Let t be a word from A^*.

2.2 Lempel-Ziv-Welch algorithm

Compression

1. Let $T_0 = \varepsilon$
2. Assume words $T_0, T_1, \ldots, T_{k-1}$ were defined, and

$$ta_0 = T_0T_1 \ldots T_{k-1}s \tag{1}$$

with s non-empty. Let T_k be the shortest prefix of s which is not among $T_0, T_1, \ldots, T_{k-1}$; there exists a unique pair consisting of a number r_k and a letter $c_k \in A \cup \{a_0\}$ such that $r_k < k$ and $T_k = T_{r_k}c_k$.

$$ta_0 = \overbrace{\underbrace{*\cdots*}_{T_{r_1}} c_1}^{T_1} \overbrace{\underbrace{*\cdots*}_{T_{r_2}} c_2}^{T_2} \; *\cdots* \; \overbrace{\underbrace{*\cdots*}_{T_{r_m}} c_m}^{T_m} \tag{2}$$

The *LZW–compression* of t is the sequence of elements $r_1, c_1, r_2, c_2, \ldots, r_m, c_m$ from $A \cup \mathbb{N}$ where m is defined by the condition $c_m = a_0$.

Decompression

1. Let $T_0 = \varepsilon$
2. Repeat $T_k = T_{r_k}c_k$ until $c_k = a_0$
3. Let $t = T_0T_1 \ldots T_{m-1}T_m$

2.3 Lempel-Ziv algorithm

Compression

1. Let $T_0 = \varepsilon$
2. Assume words $T_0, T_1, \ldots, T_{k-1}$ were defined, and

$$t = T_0 T_1 \ldots T_{k-1} s \tag{3}$$

with s non-empty. Let T_k be the longest prefix of s which is a subword of $T_0 T_1 \ldots T_{k-1}$, if such a prefix exists, otherwise let $T_k = a_j$ where a_j is the first letter of S; in the former case there exists a unique pair of numbers q_k and r_k such that $1 \leq q_k < r_k \leq |T_0 T_1 \ldots T_{k-1}|$ and $T_k = t[q_k..r_k]$; in the latter case, we define formally $q_k = r_k = -j$.

$$t = \overbrace{*\cdots*}^{T_1} * \ldots * \underbrace{t[q_k]\cdots t[r_k]}_{T_k} * \ldots * \overbrace{*\cdots*}^{T_{k-1}} \overbrace{*\cdots*}^{T_k} * \ldots * \tag{4}$$

The *LZ–compression* of t is the sequence of numbers $q_1, r_1, q_2, r_2, \ldots, q_m, r_m$ where m is such that $t = T_1 \ldots T_m$.

Decompression

1. Let $t = \varepsilon$
2. For $k = 1$ to m do : if $q_k < 0$ then let $t := t\, a_{-q_k}$ else let $t := t\, t[q_k..r_k]$

2.4 Straight-line programs

A *straight–line program* compression (in short SLP) $\mathcal{P}$ of **size** m is a sequence of assignments: $X_i := exp_i$ for $i = 1, \ldots, m$, where each X_i is a **non-terminal** and each expression exp_i is either $exp_i = a$ with $a \in A$, or $exp_i = X_j X_k$ with $k, j < i$. A straight–line program can be viewed as a context-free grammar with initial symbol X_m generating a single word $val(\mathcal{P}) = val(X_m)$ which is the decompression of the text represented in compressed form by the SLP.

2.5 Comparison of compression models

The Lempel–Ziv–Welch (resp. Lempel–Ziv) algorithm is usually called LZ78 (resp. LZ77). According to [R99] "LZ78 is less interesting [than LZ77] from the theoretical point of view, but much more interesting from the practical point of view". The size of the LZ compression of a text is smaller than the size of its LZW compression. The drawback is that the LZ compression is harder to compute.

More specifically, LZ–decompression can yield an exponential blow-up, while LZW–decompression is bounded by a quadratic growth of text size. Given an LZW–compressed text of length m, we can easily construct in time $O(m)$ an SLP of size $O(m)$ generating the decompression. Given an LZ–compressed text of length m, and of original length n, we can construct in time $O(m \log n)$ an SLP of size $O(m \log n)$ generating the same decompressed text [R03].

3 The problems

Let $t = t_1 t_2 \cdots t_n \in A^*$ be the *text* and $P = p_1 p_2 \cdots p_k$ be the *pattern* also in A^*. A size w *window* of t, in short w-*window*, is a size w subword $t_{i+1} t_{i+2} \cdots t_{i+w}$ of t; words corresponding to different values of i are considered to be different windows, even if they are equal as words; thus, there are $n - w + 1$ such windows in t. The word p is a *subsequence* of t iff there exist integers $1 \leq i_1 < i_2 < \cdots < i_k \leq n$ such that $t_{i_j} = p_j$ for $1 \leq j \leq k$. If moreover, $i_k - i_1 < w$, p is a *subsequence of t in a w-window*. A window containing p as a subsequence is said to be *minimal* if neither $t_{i+2} \cdots t_{i+w}$ nor $t_{i+1} t_{i+2} \cdots t_{i+w-1}$ contain p.

Example 1 If $t =$ "dans ville il y a vie" (a French advertisement), then "vie" is a subword and hence a subsequence of t. "vile" is neither a subword, nor a subsequence of t in a 4-window, but it is a subsequence of t in a 5-window. "ville" and "vie" are two minimal windows containing the pattern "vie". See figure 1.□

d	*a*	*n*	*s*		***v***	***i***	*l*	*l*	***e***		*i*	*l*		*y*		*a*		***v***	***i***	***e***

Fig. 1. A text with two 5-windows containing "vie" (in gray), and a single 5-window containing "vile".

Given an alphabet A, a text t on A^* and a pattern P, we consider five window problems:

- **Problem 1.** Given a compression of t and a pattern P, to decide whether pattern P is a subsequence of text t.
- **Problem 2.** Given a compression of t and a pattern P, to compute the number of minimal windows of t containing pattern P as a subsequence.
- **Problem 3.** Given a compression of t and a pattern P, to decide whether pattern P is a subsequence of a w-window of text t.
- **Problem 4.** Given a compression of t, a pattern P, and a number w, to compute the number of w-windows of t containing pattern P as a subsequence.
- **Problem 5.** Given a compression of t, a pattern P, and a number w, to compute the number of minimal windows of t which are of size at most w and which contain pattern P as a subsequence.

4 The Window Subsequence Algorithm

4.1 Auxiliary data structures we are using

From now on we will consider a text t compressed by an SLP $\mathcal{P}$ of size m. Let $|P| = k$, and let $P_1, \ldots, P_l$ (by convention $P_1 = P$) be all the different subwords of pattern P. We may note that $l = 1 + 2 + \cdots + k = k(k+1)/2 \leq k^2$.

We introduce two basic and three problem-oriented data structures.

The basic structures are two $m \times l$ arrays.

Left inclusion array. For every non-terminal X_i of program $\mathcal{P}$ and every subword P_j of pattern P, denote by $L_{i,j}$ the length of the shortest **prefix** of $val(X_i)$ containing P_j. If there is no such prefix we set $L_{i,j} = \infty$.

$$val(X_i) = \overbrace{* \ldots * \ p_\alpha * \ldots * p_{\alpha+1} * \ldots * p_{\alpha+l_j}}^{L_{i,j} \text{ symbols}} \ * \ldots * \tag{5}$$

Fig. 2. $L_{i,j}$ for $P_j = p_\alpha p_{\alpha+1} \ldots p_{\alpha+l_j}$.

Right inclusion array. For every non-terminal X_i of program $\mathcal{P}$ and every subword P_j of pattern P, denote by $R_{i,j}$ the length of the shortest **suffix** of $val(X_i)$ containing P_j. If there is no such prefix we set $R_{i,j} = \infty$.

The data structures we will use to solve the problems are the following three one-dimensional integer arrays:

Minimal windows. For every non-terminal X_i of program $\mathcal{P}$, we denote by MW_i the number of minimal windows of $val(X_i)$ containing P.

Windows of constant size. For every non-terminal X_i of program $\mathcal{P}$, we denote by FW_i the number of w-windows of $val(X_i)$ containing P.

Bounded minimal windows. For every non-terminal X_i of program $\mathcal{P}$, we denote by BMW_i the number of minimal windows of $val(X_i)$ which contain P and have size at most w.

4.2 Efficient computation of these data structures

Let us show how to efficiently compute the above five arrays.

Left inclusion array. We use structural induction over non-terminals of the SLP in order to compute the left inclusions array; the algorithm is as follows:
Basis. If $exp_i = a$, then

$$L_{i,j} = \begin{cases} 1 & \text{if } P_j = a, \\ \infty & \text{otherwise.} \end{cases}$$

Induction. If $exp_i = X_p X_q$, two cases can occur:

(i) either P_j is contained in $val(X_p)$, i.e. $L_{p,j} \neq \infty$; in that case we have $L_{i,j} = L_{p,j}$,
(ii) otherwise, let P_u be the longest prefix of P_j such that $L_{p,u} < \infty$; in this case $L_{i,j} = |val(X_p)| + L_{q,v}$ where P_v is such that $P_j = P_u P_v$.

$$\underbrace{*..*\overbrace{p_{i_u} * .. * p_{i_u+1} * .. * p_{i_u+l_u}}^{P_u \text{ subsequence}} *..*}_{val(X_p)} \overbrace{\underbrace{*..*\overbrace{p_{i_v} * .. * p_{i_v+1} * .. * p_{i_v+l_v}}^{P_v \text{ subsequence}} * ..*}_{val(X_q)}}^{L_{q,v} \text{ symbols}} \quad (6)$$

Using binary search to find P_u, the complexity of one inductive step will be $O(\log k)$, and the overall complexity will be $O(ml \log k) \leq O(mk^2 \log k)$.

Right inclusion array. Analogous to the left inclusions.

Minimal windows. We will use left and right inclusion arrays together with structural induction on the SLP structure. Let us first describe the intuitive idea. Minimal windows in $val(X_i)$ for $X_i := X_p X_q$ are of one of three types: (i) either they are entirely inside $val(X_p)$, (ii) or they are entirely inside $val(X_q)$, (iii) or they are overlapping on both $val(X_p)$ and $val(X_q)$. Type (iii) minimal windows will be called *boundary* windows (see Figure 3). To count the number of minimal windows in X_i we add the already counted numbers for X_p and X_q together with the number $B_{p,q}$ of *boundary* windows. Notice that for every decomposition $P = P_u P_v$ there is at most one boundary minimal window in which P_u is inside $val(X_p)$ and P_v is inside $val(X_q)$. Using left and right inclusion arrays we can determine decompositions of P for which such a boundary minimal window exists. However, counting must be done carefully: the same boundary window may correspond to several decompositions of P. So we run over all decompositions from $|P_u| = k-1$ to $|P_u| = 1$ and update our counter only when the following two conditions hold: 1) P_u (resp. P_v) is embedded in X_p (resp. X_q) and 2) the window is shifted from the previous successful embedding. To check these conditions, we will use a marker α in the program computing $B_{p,q}$: α will be set to 1 if we know that the next-to-be-studied window cannot be minimal. For the first and last boundary windows, we must also check that they do not contain a minimal window of type (i) or (ii), and this is also taken care of by marker α.

$$\overbrace{\underbrace{*..*\overbrace{p_1 * .. * p_2 * ... * p_l}^{P_u \text{ subsequence}} *..*}_{val(X_p)}}^{R_{p,u} \text{ symbols}} \overbrace{\underbrace{*..*\overbrace{p_{l+1} * .. * p_{l+2} * .. * p_k}^{P_v \text{ subsequence}} * ..*}_{val(X_q)}}^{L_{q,v} \text{ symbols}} \quad (7)$$

Fig. 3. A boundary window of length $R_{p,u} + L_{q,v}$ for $P = P_u P_v$, $P_u = p_1 \dots p_l$, $P_v = p_{l+1} \dots p_k$.

The algorithm is as follows.
Basis. If $exp_i = a$, then

$$MW_i = \begin{cases} 1 & \text{if } P = a, \\ 0 & \text{otherwise.} \end{cases}$$

Induction. If $exp_i = X_p X_q$, then $MW_i = MW_p + MW_q + B_{p,q}$.

$B_{p,q}$ is determined by the following FOR loop (by convention "advance" is a shorthand for $u := u'$; $v := v'$; and we write u (resp v) instead of P_u (resp. P_v)):

$B := 0$; $\alpha := 0$; $u := p_1 \dots p_{k-1}$; $v := p_k$;
IF $(R_{p,u} = R_{p,P} < \infty)$ THEN $\alpha := 1$; ENDIF
FOR $l = k-1$ TO 1 DO
  $l := l - 1$; $u' := p_1 \dots p_l$; $v' := p_{l+1} \dots p_k$;
  IF $(L_{q,v'} = L_{q,v} \wedge \infty > R_{p,u} \geq R_{p,u'})$ THEN $\alpha := 0$; advance; ENDIF
  IF $(\infty > L_{q,v'} > L_{q,v} \wedge R_{p,u} = R_{p,u'})$ THEN
    IF $\alpha \neq 1$ THEN $B := B + 1$; $\alpha := 1$; ENDIF advance; ENDIF
  IF $(\infty > L_{q,v'} > L_{q,v} \wedge \infty > R_{p,u} > R_{p,u'})$ THEN
    IF $\alpha \neq 1$ THEN $B := B + 1$; ENDIF $\alpha := 0$; advance; ENDIF
ENDFOR
IF $(L_{q,P} > L_{q,v'} \wedge \alpha \neq 1)$ THEN $B := B + 1$; ENDIF
$B_{p,q} = B$;

Thus the complexity of computing each $B_{p,q}$ is $O(k)$ and the overall complexity of computing the MW structure is $O(mk)$.

Minimal windows of size bounded by w. Computing this structure is the same as computing minimal windows. We just ignore boundary minimal windows of size more than w (*i.e.* increment B only if $(R_{p,u} + L_{q,v}) \leq w$).

Windows of constant size w. The main observation is that any w-window containing P also contains a *minimal window* containing P. Again w-windows of $val(X_i)$ (with $exp_i = X_p X_q$) containing P are (i) either entirely inside $val(X_p)$, (ii) or entirely inside $val(X_q)$, (iii) or overlapping on both $val(X_p)$ and $val(X_q)$. Thus we only need to explain how to count boundary windows. In the same way as in the previous section we run over all decompositions of P, starting from P entirely contained in $val(X_p)$ to finish with P entirely contained in $val(X_q)$. For every decomposition, using information from left and right inclusion arrays, we find a minimal window corresponding to this decomposition. In counting the number of boundary w-windows, we have to be careful, because several minimal windows can be included in the same w-window containing P as a subsequence; hence we cannot just count the number of w-windows containing a minimal window: we have to only count the *new* w-windows contributed by the current minimal window.

The number $FB_{p,q}$ of boundary w-windows is determined by a FOR loop quite similar to the previous one; we replace the statement $B := B + 1$; by a subprogram called "update" which is defined by:

IF $(R_{min} + L_{q,v}) \leq w$
THEN $B := B + R_{min} - R_{p,u}$;
ELSE IF $(R_{p,u} + L_{q,v}) \leq w$ THEN $B := B + w - (R_{p,u} + L_{q,v}) + 1$; ENDIF
ENDIF
$R_{min} := R_{p,u}$;

The number $FB_{p,q}$ boundary w-windows is defined by the following FOR loop:

$B := 0$; $l := k$; $u := P$; $R_{min} := R_{p,P}$; *// w-windows with P in val(X_p)*
IF $(R_{min} < w)$ THEN $B := B + w - R_{min}$; ENDIF
$\alpha := 0$; $u := p_1 \dots p_{k-1}$; $v := p_k$;
IF $(R_{p,u} = R_{p,P} < \infty)$ THEN $\alpha := 1$; ENDIF
FOR $l = k - 1$ TO 1 DO *// w-windows with P in val(X_p) and val(X_q)*
$l := l - 1$; $u' := p_1 \dots p_l$; $v' := p_{l+1} \dots p_k$;
IF $(L_{q,v'} = L_{q,v} \wedge \infty > R_{p,u} \geq R_{p,u'})$ THEN $\alpha := 0$; advance; ENDIF
IF $(\infty > L_{q,v'} > L_{q,v} \wedge R_{p,u} = R_{p,u'})$ THEN
IF $\alpha \neq 1$ THEN update; $\alpha := 1$; ENDIF advance; ENDIF
IF $(\infty > L_{q,v'} > L_{q,v} \wedge \infty > R_{p,u} > R_{p,u'})$ THEN
IF $\alpha \neq 1$ THEN update; ENDIF $\alpha := 0$; advance; ENDIF
ENDFOR
IF $(L_{q,P} > L_{q,v'} \wedge \alpha \neq 1)$ THEN update; ENDIF
IF $(R_{min} + L_{q,P}) \leq w$ *// w-windows with P in val(X_q)*
THEN $B := B + R_{min} - 1$;
ELSE IF $L_{q,P} \leq w$ THEN $B := B + w - L_{q,P}$; ENDIF
ENDIF
$FB_{p,q} = B$;

So we can estimate the complexity of this step by $O(k)$ and and the overall complexity of computing the FW structure is $O(mk)$.

4.3 Final algorithm and its complexity

Our structures contain answers to all five problems:

1. Pattern P is a subsequence of text t **iff** $L_{m,1} \neq \infty$ (letting $P_1 = P$),
2. The number of minimal windows of t which contain P is equal to MW_m,
3. Pattern P is a subsequence of some w-window **iff** $FW_m \neq 0$,
4. The number of w-windows containing P is equal to FW_m,
5. The number of minimal windows of size at most w and which contain P is equal to BMW_m.

So the final complexity of our algorithm in the case of compression by straight-line program is $O(mk^2 \log k)$, where m is the size of the compressed text and k is the pattern size.

Since LZW is easily converted to SLP, for LZW compression the complexity of our algorithm is $O(mk^2 \log k)$, where m is now the size of the LZW-compressed text.

For LZ compression we also can convert it to SLP. That gives as complexity $O(mk^2 \log k \log n)$. Here m is the size of the LZ-compressed text, n is the original text size and k is the pattern size.

5 Conclusions

We introduced in the present paper a new algorithm for a series of window subsequence problems. We showed that for SLP and LZW compression our algorithm is *linear* in the size of the *compressed* text. In the case of LZ compression it is only $\log n$ times worse than linear. These results show that all subsequence search problems can be done efficiently for compressed texts without unpacking.

An open question we have is the following. Is it possible to reduce the k-dependant factor in our algorithm complexity?

References

[ABF95] A. Amir, G. Benson, M. Farach, Let sleeping files lie: pattern matching in Z–compressed files, J. Comput. Syst. Sci., Vol. 52 (2) (1996), pp. 299–307.

[BKLPR02] P. Berman, M. Karpinski, L. Larmore, W. Plandowski, W. Rytter, On the Complexity of Pattern Matching for Highly Compressed Two-Dimensional Texts, Journal of Computer and Systems Science, Vol. 65 (2), (2002), pp. 332–350.

[C88] M. Crochemore, String-matching with constraints, Proc. MFCS'88, LNCS 324, Springer-Verlag, Berlin (1988), pp. 44–58.

[GKPR96] L. Gasieniec, M. Karpinski, W. Plandowski and W. Rytter. Efficient Algorithms for Lempel-Ziv Encoding (Extended Abstract), Proceedings of the 5th Scandinavian Workshop on Algorithm Theory (SWAT 1996), LNCS 1097, Springer-Verlag, Berlin (1996), pp. 392–403.

[GM02] B. Genest, A. Muscholl, Pattern Matching and Membership for Hierarchical Message Sequence Charts, In Proceedings of the 5th Latin American Symposium on Theoretical Informatics (LATIN 2002), LNCS 2286, Springer-Verlag, Berlin (2002), pp. 326–340.

[LZ77] G. Ziv, A. Lempel, A universal algorithm for sequential data compresssion, IEEE Transactions on Information Theory, Vol. 23 (3), (1977), pp. 337–343.

[LZ78] G. Ziv, A. Lempel, Compresssion of individual sequences via variable-rate coding, IEEE Transactions on Information Theory, Vol. 24, (1978), pp. 530–536.

[L05] Yu. Lifshits, On the computational complexity of embedding of compressed texts, St.Petersburg State University Diploma thesis, (2005); `http://logic.pdmi.ras.ru/~yura/en/diplomen.pdf`.

[LL05] Yu. Lifshits, M. Lohrey, Querying and Embedding Compressed Texts, to appear (2005).

[Loh04] M. Lohrey, Word problems on compressed word, ICALP 2004, Springer-Verlag, LNCS 3142, Berlin (2004), pp. 906–918.

[M02] H. Mannila, Local and Global Methods in Data Mining: Basic Techniques and open Problems, Proc. ICALP 2002, LNCS 2380, Springer-Verlag, Berlin (2002), pp. 57–68.

[MS04] N. Markey, P. Schnoebelen, A PTIME-complete matching problem for SLP-compressed words, Information Processing Letters, Vol. 90 (1), (2004), pp. 3–6.

[Ma71] Yu. Matiyasevich, Real-time recognition of the inclusion relation, Zapiski Nauchnykh Leningradskovo Otdeleniya Mat. Inst. Steklova Akad. Nauk SSSR, Vol. 20, (1971), pp. 104–114. Translated into English, Journal of Soviet Mathematics, Vol. 1, (1973), pp. 64–70; `http://logic.pdmi.ras.ru/~yumat/Journal`.

[R99] W. Rytter, Algorithms on compressed strings and arrays, Proc. SOFSEM'99, LNCS 1725, Springer-Verlag, Berlin (1999), pp. 48–65.

[R03] W. Rytter, Application of Lempel-Ziv factorization to the approximation of grammar-based compression, TCS 1-3(299) (2003), pp. 763–774.

[S71] A. Slissenko, String-matching in real time, LNCS 64, Springer-Verlag, Berlin (1978), pp. 493–496.

[W84] T. Welch, A technique for high performance data compresssion, Computer, (June 1984), pp. 8–19.