Efficient Algorithms for Lempel-Ziv Encoding

(Extended Abstract)

Leszek Gąsieniec[1] * **Marek Karpinski**[2] ** **Wojciech Plandowski**[3] *** **Wojciech Rytter**[3] †

[1] Max-Planck Institut für Informatik, Im Stadtwald, Saarbrücken D-66123, Germany.
[2] Dept. of Computer Science, University of Bonn, D-53117, Bonn, Germany.
[3] Instytut Informatyki, Uniwersytet Warszawski, Banacha 2, 02–097 Warszawa, Poland.

Abstract. We consider several basic problems for texts and show that if the input texts are given by their Lempel-Ziv codes then the problems can be solved deterministically in polynomial time in the case when the original (uncompressed) texts are of exponential size. The growing importance of massively stored information requires new approaches to **algorithms for compressed texts without decompressing**. Denote by $LZ(w)$ the version of a string w produced by **Lempel-Ziv encoding** algorithm. For given compressed strings $LZ(T)$, $LZ(P)$ we give **the first known deterministic polynomial time algorithms** to compute *compressed representations* of the set of all **occurrences of the pattern** P in T, all **periods** of T, all **palindromes** of T, and all **squares** of T. Then we consider several classical language recognition problems:

- **regular language recognition:** given $LZ(T)$ and a language L described by a regular expression, test if $T \in L$,
- **extended regular language recognition:** given $LZ(T)$ and a language L described by a LZ-compressed regular expression, test if $T \in L$, the alphabet is unary,
- **context-free language recognition:** given $LZ(T)$ and a language L described by a context-free grammar, test if $T \in L$, the alphabet is unary.

We show that the first recognition problem has a polynomial time algorithm and the other two problems are $\mathcal{NP}$*-hard.*

We show also that the LZ encoding can be computed *on-line* in **polynomial time delay and small space** (i.e. proportional to the size of the compressed text). Also the compressed representation of a pattern-matching automaton for the compressed pattern is computed in polynomial time.

---

* On leave from Institute of Informatics, Warsaw University, ul. Banacha 2, 02–097, Warszawa, Poland. WWW: `http://zaa.mimuw.edu.pl/~lechu/lechu.html`, Email:`leszek@mpi-sb.mpg.de`

** This research was partially supported by the DFG Grant KA 673/4-1, and by the ESPRIT BR Grant 7097 and the ECUS 030. Email:`marek@cs.uni-bonn.de`

*** Supported partially by the grant KBN 8T11C01208. Email:`wojtekpl@mimuw.edu.pl`

† Supported partially by the grant KBN 8T11C01208. Email:`rytter@mimuw.edu.pl`

1 Introduction

In the algorithmics of textual problems only recently the problems related to compressed objects were investigated ([2], [3], [1] and [9]). The *compressed string-matching* problem has been already investigated in [2], [3], [1], however only for the case when the text is compressed but the pattern is **not** compressed. The LZ compression (see [14]) gives a very natural way of representing a string and it is a practically successful method of text compression. Many classical algorithmic problems in the compressed setting requires efficient solution for equality testing problem. The randomized, based on finger-printing approach, version of the equality testing algorithm was already presented in [8]. In this paper we focus on **deterministic** algorithms for problems dealing with LZ-compressed strings. The first of them is the **Fully Compressed Matching Problem**:

Instance: a compressed pattern $LZ(P)$ and a compressed text $LZ(T)$

Question: does P occur in T? If "yes" then report the first occurrence, the exact number of all occurrences and a compressed set of all occurrences.

The set $\mathcal{S} = Occ(P, T)$ of all occurrences of P in T could be of exponential size with respect to $|LZ(T)|$ and by *computing* $\mathcal{S}$ we mean constructing its **compressed representation**: a data structure in which the size $|\mathcal{S}|$ and the first element of $\mathcal{S}$ are given, and we can test for the membership in $\mathcal{S}$ in time $O(n \log n)$, in case of pattern matching, and in $O(n \log n \log\log U)$ in case of palindromes and squares, where $U = |T|$.

The size of the problem is $n + m$, where $n = |LZ(T)|$ and $m = |LZ(P)|$. Assume for simplicity that $m \leq n$, and $|P| \leq |T|$ then n determines the size of the input strings and U determines the total size of the uncompressed strings. Note that in general U can be exponential with respect to n, and any algorithm which decompresses the pattern P or the text T would work in exponential time!

We consider the same version of the LZ algorithm as in [6] (this is called LZ1 in [6]). Intuitively, LZ algorithm compresses the text because it is able to discover some repeated subwords. We consider here the version of LZ algorithm without *self-referencing* but our algorithms can be extended to the general self-referential case. Assume that Σ is an underlying alphabet and let w be a string over Σ. The factorization of w is given by a decomposition: $w = c_1 f_1 c_2 \ldots f_k c_{k+1}$, where $c_1 = w[1]$ and for each $1 \leq i \leq k$ $c_i \in \Sigma$ and f_i is the longest prefix of $f_i c_{i+1} \ldots f_k c_{k+1}$ which appears in $c_1 f_1 c_2 \ldots f_{i-1} c_i$. We can identify each f_i with an interval $[p, q]$, such that $f_i = w[p..q]$ and $q \leq |c_1 f_1 c_2 \ldots f_{i-1} c_{i-1}|$. If we drop the assumption related to the last inequality then it occurs a *self-referencing* (f_i is the longest prefix which appears before but not necessarily terminates at a current position). We assume that this is not the case.

Example. The factorization of a word $aababbabbaababbabba\#$ is given by the sequence: $c_1\ f_1\ c_2\ f_2\ c_3\ f_3\ c_4\ f_4\ c_5 = a\ a\ b\ ab\ b\ abb\ a\ ababbabba\ \#$. After identifying each subword f_i with its corresponding interval we obtain the LZ encoding of the string. Hence $LZ(aababbabbababbabb\#) = a[1,1]b[1,2]b[4,6]a[2,10]\#$.

The key concepts in our algorithms are *periodicity* and *linearly-succinct* representations of exponentially many periods. The concept of periodicity appears in many advanced string algorithms, it is naturally related to LZ compression, since the high compression ratio is achieved when there are many repetitions in the text and repetitions are closely related to the periodicity.
Denote $Periods(w) = \{p : p \text{ is a period of } w\}$. A set of integers forming an arithmetic progression is called here *linear*. We say that a set of positive integers from $[1 \dots U]$ is *linearly-succinct* iff it can be decomposed in at most $\lfloor \log_2(U) \rfloor + 1$ linear sets. The following lemma was shown in [9].

Lemma 1. LINEARLY-SUCCINCT SETS LEMMA
The set $Periods(w)$ is linearly-succinct.

In this paper the terms *linearly-succinct* and *compressed* representations have different meanings, however each *linearly-succinct* set has a *compressed* representation.
Observation. The set of all occurrences of a given pattern in a compressed text T is not necessarily linearly-succinct, even if the pattern is a single letter.

In [9] and [12] the compressed strings were considered in terms of context-free grammars (grammars, in short) generating single words. We can prove:

Lemma 2. *For each string w given in LZ-compressed form we can construct a context-free grammar generating w of size $O(n^2)$, where $n = |LZ(w)|$.*

Proof. We omit the proof. □

In this paper we consider more compact and more useful abstraction of *LZ*-encodings in terms of *composition systems*. The *composition system approach* reduces the complexity considerably compared with context-free grammars. The *composition systems* (introduced here) are variations of straight line programs and context-free grammars. Introduce the set $VAR(\mathcal{S})$ of *variables* of a composition system $\mathcal{S}$. The variables correspond to the subwords f_i (to intervals $[i, j]$ in the LZ encoding of a given word w). The value of each variable is a string over Σ. Denote by $Y_{[i]}$ and $Z^{[i]}$ the prefix of length i of Y and the suffix of length i of Z. Denote by $\cdot$ the operation of concatenation. The **composition system** $\mathcal{S}$ is a sequence of *composition rules* of the form: $X = Y^{[i]} \cdot Z_{[j]}$, $X = Y \cdot Z$ or $X = a$, where $a \in \Sigma$. Each variable appears exactly once on the left side of a composition rule. It is called *the rule* for this variable. The rule for a variable occurs before all rules in which this variable is on their right sides. The variables whose compositions are of the form $X = a$ are called *atomic*. The values of atomic variables are the constants which appear on the right sides. The value of the last variable of the composition system $\mathcal{S}$ is the value of $\mathcal{S}$ and is denoted by $val(\mathcal{S})$.

Example. We have $val(\mathcal{S}) = val(G) = abbababababba$ in the following composition system $\mathcal{S}$:

$A = a$; $B = b$; $C = A \cdot B$; $D = B \cdot C$; $E = C \cdot D$; $F = D^{[2]} \cdot E_{[4]}$; $G = E \cdot F$.

Lemma 3. *For each string w given in LZ-compressed form and a subword v of w we can construct a composition system generating v of size $O(n \cdot \log(n))$, where $n = |LZ(w)|$, for each variable A the word $val(A)$ is a subword of w.*

Proof. The proof will be given in the full version of the paper. □

Fact 4 *Testing for an occurrence of pattern $\mathcal{P}$ in text $\mathcal{T}$ can be reduced to the following problem:*
• for two composition systems $\mathcal{P}$, $\mathcal{T}$ decide whether there is a variable X in the system $\mathcal{T}$ such that
• $val(\mathcal{P})$ is a subword of $val(X)$.

For a given string w generated by a composition system $\mathcal{S}$ we consider the **bracket structure** of w with respect to $\mathcal{S}$. The bracket structure appears if we replace each composition rule $X = Y^{[i]} \cdot Z_{[j]}$ by $X = (\ Y^{[i]} \cdot Z_{[j]}\ )$, where (and) are new terminal symbols. In this way inside the text w many pairs of brackets will appear, each of them associated with a variable. The brackets define the family $\mathcal{F}$ of intervals in w, each interval is associated with a variable, consider only intervals which are decomposed into two intervals (according to a composition of the composition system). The point of the decomposition is called the **cut**. Our compressed representations of objects in w are related to the bracket structure, for example the set of all occurrences of the pattern is represented as a set of occurrences for each interval in $\mathcal{F}$. An occurrence is **associated with** an interval τ iff τ is the smallest interval in $\mathcal{F}$ containing the pattern P and the pattern *overlaps* the cut of τ. The first crucial point is that the number of *essentially different* intervals in $\mathcal{F}$ is only linear, since they correspond to n variables, though $|\mathcal{F}|$ can be exponential. The second crucial point is that the set of occurrences *associated with* a given interval in $\mathcal{F}$ forms a single arithmetic progression. Moreover for each position i it is easy to find an interval with which i is (possibly) *associated* by traversing *top-down* the generation tree of w. Then we test if i is in an arithmetic progression of positions *associated with* a given interval. The height of the tree is at most n. This approach gives the membership algorithms working in linear time with respect to n.

2 Fully Compressed String-Matching

Our auxiliary problem is that of checking if a part of the pattern P occurs at a given position i in T. The **Compressed Equality Testing** problem is described here as follows.

Instance: a compressed pattern $LZ(P)$, a compressed text $LZ(T)$ and integers i, j, i', j'

Question: does $P[i..j] = T[i'..j']$? If "no" then find the first **mismatch**.

Assume we have two composition systems $\mathcal{S}_1$, $\mathcal{S}_2$ with n_1, n_2 variables, respectively. The key point of our algorithm is to consider relations between some parts of the words which are values of variables in these composition systems. For the

convenience we sometimes identify the name of the variable with its value.
The main object in our algorithm is an information that two parts of some variables are equal: $A[p..q] = B[p'..q']$, where $p = 1$ or $q = |A|$ or $p' = 1$ or $q' = |B|$. Such information is stored in objects called here the *equality-items* (*items*, in short). There are three types of *items*:

1. *overlap items*: $OV(A, B, i)$ means that $B[1..i]$ is a suffix of A (in other words $A^{[i]} = B_{[i]}$);
2. *suffix items*: $SU(A, B, i, k)$ means that $A[i..k]$ is a suffix of B;
3. *prefix items*: $PR(A, B, i, k)$ means that $A[i..k]$ is a prefix of B.

We distinguish also another type of items: *subword items*. These are the prefix (suffix) items in the case when the whole word B is considered as its prefix (suffix). The items will be denoted by small greek letters α, β, γ, possibly with subscripts. The sets of items will be denoted by capital greek letters. Each item corresponds to equality of two subwords, the item is **valid** iff this equality is satisfied. The set Γ of items is valid iff each item in Γ is valid. The *size* of an item is the length of the text which "takes part" in the equality. If the item is of type $OV(A, B, i)$ then the equality concerns the prefix $B_{[i]}$ and the suffix $A^{[i]}$, both of length i. Hence $size(OV(A, B, i)) = i$. The sizes of other types of items are defined similarly. The items of size one are called **atomic items**. The validity of atomic items is rather simple. Two sets Γ_1 and Γ_2 of items are equivalent (we write $\Gamma_1 \equiv \Gamma_2$) iff validity of Γ_1 is equivalent to validity of Γ_2. The basic operation in our equality-test algorithm is $SPLIT(\alpha)$, where α is an item. The value of this operation is a set of one or two items: α is split into "smaller" items. The operation satisfies: $\{\alpha\} \equiv SPLIT(\alpha)$. The operation $SPLIT$ can be defined formally in a similar way as in [12]. We describe only how overlap items are split, other types of items are split similarly. Assume $A, B \in VAR(\mathcal{S}_1) \cup VAR(\mathcal{S}_2)$. Assume our item α is $OV(A, B, i)$, (which means $A^{[i]} = B_{[i]}$) and the composition rule related to A is: $A = C^{[p]} \cdot D_{[q]}$.

Case 1: $i \le q$. Now $A^{[i]} = B_{[i]}$ corresponds to $(D_{[q]})^{[i]} = B_{[i]}$. Thus $SPLIT(\alpha) = \{PR(D, B, q - i + 1, q)\}$.

Case 2: $i > q$. Now $A^{[i]} = B_{[i]}$ is equivalent to $C^{[i-q]} = B_{[i-q]}$ and $(B_{[i]})^{[q]} = D_{[q]}$, and $SPLIT(\alpha) = \{OV(C, B, i - q), PR(B, D, i - q + 1, i)\}$.

Assume that in each $SPLIT$ exactly one variable (the longer one) is *decomposed*. The variables participating in $SPLIT$ are different (one from system $\mathcal{S}_1$, another from $\mathcal{S}_2$). Assume X, Y are the last variables in composition systems $\mathcal{S}_1$, $\mathcal{S}_2$, respectively. The equality-test checks if $val(X) = val(Y)$.

The item $\alpha_0 = OV(X, Y, |X|)$ is called the *starting item*. Hence, if $|X| = |Y|$, then the equality-test problem is reduced to the validity problem for the starting item. The structure of the algorithm is:

Initially: $\Gamma = \{\alpha_0\}$
invariant1: $\Gamma \equiv \{\alpha_0\}$.
invariant2: $|\Gamma|$ is polynomial.
Finally: Γ consists only
of *atomic* items.

Assume Δ is a set of overlap items. We use the operation $Compact(\Delta)$. Essentially this operation works similarly as in [12]. The operation removes from Δ some number of overlap items and possibly inserts other overlap items (related to the same variables as removed items).
Assume that for each atomic item α holds the equality $SPLIT(\alpha) = \{\alpha\}$. Let $\Delta_{A,B}$ be the set of overlap items of the type $OV(A, B, *)$. This set is represented by the largest overlap and set of periods in a prefix of B, since other overlaps correspond to shifts of B on itself. However such set of periods can be represented by a set of $\log U + 1$ size due to Lemma 1. The overlaps correspond to periods and due to Lemma 1 we can choose only $\log U + 1$ number of items of a given type $OV(A, B, *)$. We have quadratic number of pairs of variables, hence the upper bound for $|Compact(\Delta)|$ is $O((n_1 + n_2)^2 \log U)$.

ALGORITHM *EQUALITY_TEST* ;
$\Gamma := \{ \alpha_0 \}$;
while Γ contains a non-atomic item **do**
  $\Gamma' := \emptyset$;
  for each $\alpha \in \Gamma$ **do** $\Gamma' := \Gamma' \cup SPLIT(\alpha)$;
  $\Gamma := Compact(\Gamma')$;
if all items in Γ are valid **then return** *true*
  else return *false*

The operation *Compact* satisfies:

$$\Delta \equiv Compact(\Delta) \text{ and } |Compact(\Delta)| \leq c \cdot (n_1 + n_2)^2 \log U,$$

for a constant c.
If $\Gamma = \Delta_1 \cup \Delta_2$, where Δ_1 is the set of overlap items in Γ and Δ_2 is the set other items then define

$$Compact(\Gamma) = Compact(\Delta_1) \cup \Delta_2.$$

It can be proved that $|\Gamma| \leq c \cdot (n_1 + n_2)^3 \log U$ and this implies the following theorem (the details are omitted).

Theorem 5. *The Compressed Equality Testing problem can be solved in time* $O((n \log n)^4 \log^2 U)$.

We will now describe how to apply the *Compressed Equality Testing* in pattern matching algorithm. Denote by $Eq(m)$ the time complexity of equality test algorithm for two compressed texts of size m. Denote $ArithProg(i, p, k) = \{i, i + p, i + 2p, \ldots, i + kp\}$, so it is an arithmetic progression of length $k + 1$.

Its description is given by numbers i, p, k written in binary representation. Our pattern matching algorithm deals with a polynomial number of arithmetic progressions representing periods or overlaps.
Denote by $Solution(p, V, W)$ any position $i \in V$ such that $i + j = p$ for some $j \in W$. In case there is no such position i then $Solution(p, V, W) = 0$. Moreover let $\#Solution(p, V, W)$ be the number of such is.

Lemma 6. APPLICATION OF EUCLID'S ALGORITHM
Assume that two linear sets $V, W \subseteq [1 \ldots U]$ are given by their descriptions. Then for a given number $c \in [1 \ldots U]$ we can compute $Solution(c, V, W)$ and $\#Solution(c, V, W)$ using $\log^2(U)$ arithmetic operations.

We show only how to find an occurrence of the pattern, the number of occurrences can be computed by applying Lemma 6 during the second phase of the algorithm. Let us fix the pattern $P = val(\mathcal{P})$. Let $val(X_i)$ be a string of length K and j be any position in this string. Define $Pre(j, i)$ to be the lengths of subwords of $val(X_i)$ that end at position j in $val(X_i)$ and that are prefixes of P. Similarly, denote by $Suf(j, i)$ the lengths of subwords of $val(X_i)$ that begin at position j in $val(X_i)$ and that are suffixes of P. Formally: $Pre(j, i) = \{1 \le i \le j \ : \ X_i[j - i + 1..j]$ is a a prefix of P $\}$.
$Suf(j, i) = \{j \le i \le K \ : \ X_i[j..j + i - 1]$ is a suffix of P $\}$. Observe that these tables depend on the pattern P, however it is convenient to assume further that P is fixed.

Lemma 7. *Let $X_i = X_j \cdot X_k$, then P occurs in X_i iff P occurs in X_j or P occurs in X_k, or $Solution(|P|, Pre(|X_j|, j), Suf(1, X_k)) \neq 0$.*

Let $X_1, X_2, \ldots, X_m$ be a sequence of variables that appear in consecutive rules of the composition system $\mathcal{T}$ defining the text T. By Lemma 3 $m = O(n \log n)$.

Let k be the first position in $Prefs(j, i)$, then all the other positions in $Prefs(j, i)$ are of the form $k + p'$, where p' is a period of $P[1..k]$. Hence Lemma 1 implies directly the following fact.

Lemma 8. *The sets $Suffs(j, i)$ and $Prefs(j, i)$ are linearly-succinct.*

Let the composition rule for a variable X_k be of the form $X_k = X_i^{[s]} \cdot X_{j[t]}$. Then for each position b in the word $val(X_k)$ denote by $Pred[b, X_k]$ the position $|val(X_i)| - s + b$ in the word X_i if $1 \le b \le s$ and the position $b - s$ in the word X_j if $b > s$. These positions are associated with corresponding variables. If the composition rule for a variable X_k is of the form $X_k = a$ then $Pred[b, X_k]$ is undefined. The function $Pred$ (predecessor) defines a partial order "to be a predecessor" between pairs (position in $val(X_k)$, variable X_k).
We are now able to give a sketch of the whole structure of the algorithm. In the first phase of the algorithm in each word $val(X_k)$ at most $2m$ positions are being distinguished. They are called **finger** and are denoted by $Finger(X_k)$. They correspond to *informers* in [6]. The positions in $Finger(X_k)$ are those predecessors of end-positions of variables that are in the word $val(X_k)$. Clearly,

for a fixed k, in the set $Finger(X_k)$ there are at most two predecessors of end-positions of one variable, so that there are at most $2m$ positions in the set $Finger(X_k)$.

Lemma 9. *The sets $Finger(X_k)$ can be computed in $O(n^2)$ time.*

```
ALGORITHM SECOND_PHASE ;
  for i := 1 to n do
      for each informer b in X_i do
          compute Prefs(b, i) and Suffs(b, i);
      test for an occurrence using Lemma 6 and Lemma 7
```

The values of $Prefs(b,i)$ and $Suffs(b,i)$ are computed as follows. Let c be the cut of $val(X_k)$. First, the values of $Prefs(c,k)$ and $Suffs(c,k)$ are computed. Then for each arithmetic sequence in those sets we compute the set of **maximal pattern prefixes** which start in the positions of the sequence. As it is proved in [9] such set can be represented in constant space and it is enough to compute such prefixes only for three positions of the sequence. Given a position i in $val(X_k)$ the length of the maximal pattern prefix which starts at i in $val(X_k)$ can be computed by **binary search** using $\log|val(X_k)|$ times the **equality test** procedure. Then the sets $Prefs(b,k)$ and $Suffs(b,k)$ are computed. For each finger b we put to $Prefs(b,k)$, for $b \le c$ the prefixes from $Prefs(Pred[b,k])$ which are shorter than b, and, for $b > c$, the set $Prefs(Pred[b-c,k])$ and the parts of sequences from $Prefs(c,k)$ for which the maximal prefixes extend over b. Similarly we compute the sets $Suffs(b,k)$. The time complexity is dominated by computing maximal prefixes. Computing one prefix is done in $O(Eq(m)\log U)$ time where m is the size of the composition system generated from the LZ-code. There are at most $m\log U$ prefixes to compute. This gives the total time complexity $O(m\log^2 U \cdot Eq(m))$. Since $m = O(n\log n)$, we have:

Theorem 10. *The Fully Compressed Matching Problem can be solved in time $O(n\log n\log^2 U \cdot Eq(n\log n))$.*

As a side effect of our pattern-matching algorithm we can compute the set of all periods for strings with short description.

Theorem 11. *Assume $\mathcal{P}$ is a compressed pattern. Then the compressed representation of the set $Periods(\mathcal{P})$ can be computed in $O(n\log n\log^2 U \cdot Eq(n\log n))$ time. The representation consists (in this case) of $\log U$ number of linear sets.*

Proof. Use our string-matching algorithm with the pattern $\mathcal{P} = val(\mathcal{S})$ and the text $\mathcal{T} = val(\mathcal{S})$ ignoring the occurrence of the pattern at position 1. As a side effect we compute all suffixes of $\mathcal{T}$ which are prefixes of $\mathcal{P}$. This determines all periods. □

3 Compressed Palindromes and Squares

Our algorithms for compressed **palindromes** and **squares** use ideas from [4]: palindromes are searched using **periodicities** implied by sequences of many palindromes which are *close to each other* and searching of squares is reduced to multiple application of **pattern-matching**.

First we consider a data structure for palindromes, consider only even length palindromes (the algorithms for odd length palindromes are similar). Let $w[i..j]$ be a subword of w which is a palindrome. This palindrome is **centered in position** $(i+j)/2$ in w and its **radius** is $(j-i)/2$. Let $Radius[1..|w|]$ be an array of radii of palindromes, this means that the entry $Radius[i]$ contains the maximal radius of a palindrome centered in i. The array $Radius$ is a representation of all even length palindromes inside a word. Indeed, $w[i..j]$ is an even palindrome iff (i+j) is odd and $Radius[(i+j-1)/2] > (j-i-1)/2$. A compressed representation of the array $Radius$ is called here a **pal-structure**. A palindrome which is a prefix (suffix) of a word is called *initial* (*final*) palindrome. Denote by $InitPal(w,i)$ ($FinPal(w,i)$) the set of initial palindromes in $w[i+1..|w|]$ (final palindromes in $w[1..i]$). We say that a palindrome **covers** a given position i iff i is inside this palindrome. In other words, the palindrome of radius r centered in j covers i iff $j-r < i \leq j+r$.

Lemma 12. *The sets $InitPal(w,i)$ and $FinPal(w,i)$ are both linearly-succinct. Moreover, if $i_1 > i_2 > \ldots > i_k$ is a set of positions in $InitPal(w,i)$, then the sequence $\{i_s\}$ can be divided in $O(\log|w|)$ blocks of arithmetic sequences.*

The **pal-structure** consists of sequences of numbers which are assigned to variables in a composition system. Let A be a variable and let the rule for A be $A = E^{[r]}F_{[s]}$. Denote by $LeftPal(A)$ ($RightPal(A)$) the set of centers of palindromes which cover r and that are not in $FinPal(A,|A|)$ ($InitPal(A,1)$) and that are to the left (right) of r. Formally, $LeftPal(A) = \{j : j < r < j + Radius[j], j \notin FinPal(A,|A|)\}$. Clearly, $LeftPal(A) \subseteq FinPal(E^{[r]})$ and $RightPal(A) \subseteq \{r+i : i \in InitPal(F_{[s]})\}$. The pal-structure consists of the sets $LeftPal(A)$, $RightPal(A)$, for each variable A. Each arithmetic sequence in $LeftPal(A)$ and $RightPal(A)$ is equipped with the values of $Radius$ for the first, last and some other element of the sequence.

Theorem 13.
(A) The compressed representation of all palindromes in the compressed text can be computed in $O(n \log n \log^2 U \cdot Eq(n \log n))$ time.
(B) Given a position i and the compressed representation of all palindromes, the value of $Radius[i]$ can be computed in $O(n \log n \log\log U)$ time.

Proof. To appear in the full version of the paper. □

Theorem 14. *The compressed representation of the set of all squares in the compressed text can be computed in $O(n^2 \log^2 n \log^3 U \cdot Eq(n \log n))$ time.*

Proof. To appear in the full version of the paper. □

4 Compressed Language Recognition Problems

For a given regular expression describing the language R and a context-free grammar G generating a language L we can construct a grammar generating the language $L \cap R$. The construction proves also the following theorem (we omit details in this version).

Theorem 15. *The following problem has a polynomial time algorithm: given* $LZ(T)$ *and a language* L *described by a regular expression, test if* $T \in L$.

Proof. First, the encoding $LZ(T)$ is transformed into a context-free grammar G which generates the language $L' = \{T\}$. Then a context-free grammar for the language $L' \cap L$ is constructed. Finally, the language $L' \cap L$ is tested for the emptiness. All steps can be accomplished in polynomial time. This completes the proof. □

The problem becomes *computationally infeasible* if we allow exponentiation in regular expressions. Such expressions are called here *extended regular expressions*. Consequently, the problem is $\mathcal{NP}$-hard for context-free grammars.

Theorem 16. *The following two problems are* $\mathcal{NP}$*-hard:*
(1) given $LZ(T)$ *and a language* L *described by an extended regular expression, test if* $T \in L$*, the expression does not contain empty string and the alphabet is unary,*
(2) given $LZ(T)$ *and a language* L *described by a context-free grammar, test if* $T \in L$*, the alphabet is unary.*

Proof. Part (2) is a consequence of part (1). The proof of part (1) is a reduction from the SUBSET SUM problem defined as follows:

Input instance: Finite set $A = \{a_1, a_2, \ldots, a_n\}$ of integers and an integer K. The size of the input is the number of bits needed for the description.

Question: Is there a subset $A' \subseteq A$ such that the sum of the elements in A' is exactly K?

The problem SUBSET SUM is $\mathcal{NP}$-complete, see [7]. The LZ code for the string $P = 1^K$ is of size $O(\log K)$. The SUBSET SUM problem is reduced to the membership:

$$P \in (1^{a_1} \cup \varepsilon) \cdot (1^{a_2} \cup \varepsilon) \cdots (1^{a_n} \cup \varepsilon).$$

The empty string ε can be easily eliminated by replacing each ε and 1^{a_i} by a single letter 1 and by 1^{a_i+1}, respectively, and concatenating P with 1^n. This completes the proof. □

5 Two Applications of Our Compressed Representation

At the end we show two applications of our compressed representations.

Lemma 17. *Assume* $a \in \Sigma$ *and we have a compressed representation of the set of occurrences of the pattern* P *in* T*, then we can compute the compressed set of all occurrences of* Pa *in* $O(n^2 \log^2 n)$ *time.*

Proof. Recall, that the compressed representation of a set of occurrences of a pattern P in T is a composition system representing T and additionally for each variable X in the system one arithmetic sequence of those occurrences of P in X which cover the cut of X. We show how to modify the sequences to obtain the sequences for the pattern Pa. Let $a_1 < a_2 \ldots < a_k$ be such a sequence. Then we check whether the occurrences of P at a_1 and a_k can be extended to occurrences of Pa. This can be done in $O(m)$ time where m is the size of the composition system for P ($m = O(n \log n)$). If the occurrence of P at a_1 cannot be extended by a then no occurrence of P at a_i, for $i < k$, can be extended by a. If the occurrence of P at a_1 can be extended by a then all occurrences of P at a_i, for $i < k$, can be extended by a. This allows to construct a part of the sequence for Pa. To obtain the final sequence we have to check whether there is an occurrence of Pa which sticks to the cut of a variable. This can be done in $O(m)$ time. This completes the proof. □

Theorem 18. *For a given text T, its code $LZ(T)$ can be computed on-line with $O(n^2 \log^2 n)$ delay using $O(n \log n)$ space.*

Proof. It is enough to show how to compute in the claimed complexities a code $LZ(Sa)$ given a code $LZ(S)$ for a string S. Let $c_1 f_1 \ldots c_k f_k$ be the LZ-factorization of S. Then the factorization for Sa is either $c_1 f_1 \ldots c_k f'_k$ where $f'_k = f_k a$ (if $f_k a$ occurs inside S), or $c_1 f_1 \ldots c_k f_k a$ (if $f_k a$ does not occur inside S). If we keep the compressed representation of the set of occurrences of the last factor f_k inside S then by Lemma 17 the code $LZ(Sa)$ and the set of occurrences of the last LZ-factor of Sa can be computed in $O(n^2 \log^2 n)$ time and $O(n \log n)$ space. This completes the proof. □

We can compute the compressed representations of the table $Next$ from the Knuth-Morris-Pratt algorithm, see [5], and of the pattern-matching automaton. This gives another application of our data structure.

Theorem 19. *The compressed representation $\mathcal{A}$ of the pattern-matching automaton (of size polynomial on n) for a compressed pattern P can be constructed in polynomial time in $LZ(P)$. The constructed automaton works with the delay $O(n \log n \cdot \log U)$.*

6 Concluding Remarks

We proved that the membership problem for extended regular expressions and for context-free grammars are $\mathcal{NP}$-hard if an input word is compressed. An open problem is the membership in $\mathcal{NP}$. The same problem for regular grammars is in $\mathcal{P}$. The important open problems are the complexity of the membership problem for deterministic context-free grammars. If the first problem is in $\mathcal{P}$ then parsing of compressed programs can be done polynomially with respect to the size of the compressed program.

Our method yields the first polynomial time algorithm for the **LZ Fully Compressed Matching Problem** and other classical problems for texts. An interesting open problem remains on improving running time and storage requirements of our algorithms.
This is the first approach on polynomial-time deterministic algorithms for *fully* compressed versions of problems on texts.

Our algorithms mostly make sense for highly compressed texts, an open problem is to adjust them to the case when the compression is small. Of course if there is no compression then the standard algorithms (for noncompressed texts) work, but in the case of small compression it seems that a hybrid of these standard algorithms and algorithms presented here would be more efficient.

References

1. A. Amir, G. Benson and M. Farach, *Let sleeping files lie: pattern-matching in Z-compressed files,* in *SODA'94*.
2. A. Amir, G. Benson, *Efficient two dimensional compressed matching, Proc. of the 2nd IEEE Data Compression Conference* 279-288 (1992).
3. A. Amir, G. Benson and M. Farach, *Optimal two-dimensional compressed matching,* in *ICALP'94*.
4. A. Apostolico, D. Breslauer, Z. Galil, Optimal parallel algorithms for periods, palindromes and squares, in *ICALP'92*, pp. 296-307.
5. M. Crochemore and W. Rytter, *Text Algorithms,* Oxford University Press, New York (1994).
6. M. Farach and M. Thorup, *String matching in Lempel-Ziv compressed strings,* in STOC'95, pp. 703-712.
7. M.R. Garey and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman (1979).
8. L. Gąsieniec, M. Karpiński, W. Plandowski and W. Rytter, *Randomized Efficient Algorithms for Compressed Strings: the Finger-Print Approach,* to appear in proceedings of the 7th Combinatorial Pattern Matching, Laguna Beach (1996).
9. M. Karpinski, W. Rytter and A. Shinohara, *Pattern-matching for strings with short description,* in *Combinatorial Pattern Matching*, 1995.
10. D. Knuth, *The Art of Computing, Vol. II: Seminumerical Algorithms. Second edition.* Addison-Wesley, 1981.
11. A. Lempel and J. Ziv, *On the complexity of finite sequences, IEEE Trans. on Inf. Theory* 22, 75-81 (1976).
12. W. Plandowski, *Testing equivalence of morphisms on context-free languages,* ESA'94, Lecture Notes in Computer Science 855, Springer-Verlag, 460–470 (1994).
13. J. Storer, *Data compression: methods and theory,* Computer Science Press, Rockville, Maryland, 1988.
14. J. Ziv and A. Lempel, *A universal algorithm for sequential data compression, IEEE Trans. on Inf. Theory* vo. IT–23(3), 337–343, 1977.