The Fully Compressed String Matching for Lempel-Ziv Encoding

Marek Karpinski [*]

Wojciech Plandowski [†]

Wojciech Rytter [‡]

Abstract

The growing importance of massively stored information requires new approaches to efficient algorithms on texts represented in a compressed form. We consider here the *string-matching* problem in the compressed setting. This problem has been already investigated in [2], [3], [1]. A rather theoretical type of compression was considered in [8]. In this paper we consider a practically important compression algorithm of Lempel and Ziv (*LZ algorithm*, in short). Denote by $LZ(w)$ the compressed version of a given string w using the LZ algorithm. The ***Fully Compressed Matching Problem*** is that of deciding if the pattern P occurs in a text T, given only $LZ(P)$ and $LZ(T)$, without decompressing the pattern and the text. The first occurrence is reported, if there is any. Let m and n denote the sizes of $LZ(P)$ and $LZ(T)$, and M, N be the sizes of uncompressed strings P and T, respectively.

In this paper we design the first polynomial time (with respect to n and m) algorithm for the *Fully Compressed Matching Problem.* Note that in generall N, M are exponential with respect to n and m, and any algorithm which explicitely decompresses the pattern P or the text T would work in exponential time! In particular the algorithm given in [5] works in this situation in exponential time with respect to m (in this algorithm the uncompressed pattern is a part of the input). The situations when both objects participating in the *string-matching* are compressed (we deal with compressed patterns) are also practically important, for example in genetics and molecular biology (where uncompressed patterns are extremely long) or when we search for one compressed file in another compressed file. We introduce a new technique of succinct representations for long string periods and overlaps.

[*]Dept. of Computer Science, University of Bonn, 53117 Bonn. This research was partially supported by DFG Grant KA 673/4-1, and by the ESPRIT BR Grant 7097 and ECUS 030. Email:`marek@cs.uni-bonn.de`

[†]Institute of Informatics, Warsaw University, 02–097 Warszawa. Partially supported by the KBN Grant. Email:`wojtekp@mimuw.edu.pl`

[‡]Institute of Informatics, Warsaw University, 02–097 Warszawa. Supported partially by the DFG Grant Bo 56/142-1. Email:`rytter@mimuw.edu.pl`

1 Introduction

Gathering and storage of masses of data is closely related to data compression. It is of practical interest to be able to answer some type of queries without data decompression and with the efficiency proportional to the size of compressed objects. Compression is a kind of succinct representation. The complexity of succinctly represented graphs was already investigated in many papers. However in the algorithmics of textual problems only recently the problems related to compressed objects were investigated ([2], [3], [1] and [8]). The *compressed matching problem* was investigated in [5], where the Lempel-Ziv (**LZ**) compression was considered. The LZ compression (see [14]) gives a very natural way of representing a string. In this paper we consider the **Fully Compressed Matching Problem**:

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

Question: does P occurs in T ? If "yes" then report the first occurrence.

Denote $m = |LZ(P)|$ and $n = |LZ(T)|$. Let $M = |P|$ and $N = |T|$. It can happen that $M = \Omega(2^{c \cdot m})$, $N = \Omega(2^{c \cdot n})$. Hence the pattern and the text are too long to be written *explicitly.* Fortunately, each position can be written with only linear number of bits. Our main result is the following theorem.

Theorem 1.1
The Fully Compressed Matching Problem can be solved in polynomial time with respect to $m + n$.

The algorithm from the theorem above is deterministic, it is polynomial time but the degree of the polynomial is still high ($O((n + m)^6)$). (Of course it is much more efficient than the algorithm with *explicit decompressing* which works in exponential time.) The key idea is the succinct representation of sets of exponentially many periods and overlaps. 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', where $j - i = j' - i' \geq 0$.

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

1.1 The Lempel-Ziv compression.

There is a large number of possible variations of the LZ algorithm. We consider the same version as in [5] (this is called LZ1 in [5]). Intuitively, LZ algorithm compresses the text because it is able to discover some repeated subwords. We consider the version of LZ algorithm without *self-referencing.* 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 there is no such situation.

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

1.2 Composition systems

We introduce some useful abstraction of the LZ encoding. 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. Let $\cdot$ denote 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. 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 $\mathcal{S}$ (denoted $val(\mathcal{S})$) is the value of $\mathcal{S}$.

Example Consider the following composition system $\mathcal{S}$:
$A = a$; $B = b$; $C = A \cdot B$; $D = B \cdot C$; $E = C \cdot D$; $F = D^{[2]} \cdot E_{[4]}$; $G = E \cdot F$.

We have here $val(\mathcal{S}) = val(G) = abbababababba$.

We say that the systems $\mathcal{S}_1$, $\mathcal{S}_2$ are **equivalent** iff $val(\mathcal{S}_1) = val(\mathcal{S}_2)$. Assume we are given the code $LZ(w)$ of the word w. Then we can reconstruct in polynomial time the factorization $c_1\ f_1\ c_2\ f_2 \ldots\ c_p\ f_p\ c_{p+1}$ of w corresponding to this encoding. For each f_k we can compute in polynomial time the integers i, j such that $f_k = w[i..j]$. Using this information we can easily construct the composition system corresponding to any subinterval of w. If a subinterval corresponds to composition of more than two factors f_i then we use a method similar to the transformation of a grammar to a normal form. We can prove the following fact.

Fact 1.2
(1) The compressed equality test can be reduced in polynomial time to the equivalence problem for two composition systems.
(2) The Fully Compressed Matching Problem 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 $val(X)$.

1.3 The structure of periods in long strings

The concept of periodicity appears in many advanced string algorithms, it is intuitively 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.
A nonnegative integer p is a *period* of a nonempty string w iff $w[i] = w[i-p]$, whenever both sides are defined. Hence $p = |w|$ and $p = 0$ are considered to be periods. 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 $[1 \ldots N]$ is *succinct* w.r.t. N iff it can be decomposed in at most $\lfloor \log_2(N) \rfloor + 1$ linear sets. The following lemma was shown in [8].

Lemma 1.3 (succinct sets lemma)
The set $Periods(w)$ is succinct w.r.t. $|w|$.

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. Our pattern matching algorithm deals with a polynomial number of arithmetic progressions representing periods or overlaps.

Denote by $Solution(p,U,W)$ any position $i \in U$ such that $i + j = p$ for some $j \in W$. If there is no such position i then $Solution(p,U,W) = 0$.

Lemma 1.4
(application of Euclid's algorithm)

Assume that two linear sets $U, W \subseteq [1 \ldots N]$ are given by their descriptions. Then for a given number $c \in [1 \ldots N]$ we can compute $Solution(c,U,W)$ in polynomial time with respect to $\log(N)$.

2 The Compressed Equality-Test Algorithm

According to Fact 1.2 the Compressed Equality-Test problem is reduced to the equivalence problem for two composition systems. We show that the latter problem can be done in polynomial time.
Assume we have two composition systems $\mathcal{S}_1, \mathcal{S}_2$ with n_1, n_2 variables. The key point of our algorithm is to consider relations between some parts of the words which are values of variables in these systems. We 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.

Observe that each overlap item is a special type of a prefix item, however the introduction of overlap items plays the crucial role in our algorithm.

There is 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 letters $\alpha, \gamma\, \beta$, possibly with subscripts. The sets of items will be denoted by capital greek letters.

Each item corresponds to equality of two subwords. An 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 $OV(A, B, i)$ 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.

Fact 2.1 *The validity of atomic items can be tested in polynomial time.*

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 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: $i \leq q$.
In this case $A^{[i]} = B_{[i]}$ is equivalent to $D^{[i]} = B_{[i]}$. Hence $SPLIT(\alpha) = \{OV(D, B, i)\}$.

Case 2: $i > q$. In this case $A^{[i]} = B_{[i]}$ is equivalent to

$$C^{[i-q]} = B_{[i-q]} \text{ and } B^{[i-q+1..i]} = D_{[i-q]}.$$

Hence
$SPLIT(\alpha) = \{OV(C, B, i - q),$
$PR(B, D, i - q + 1, i)\}$, see Figure 1.

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 iff $val(X) = val(Y)$.

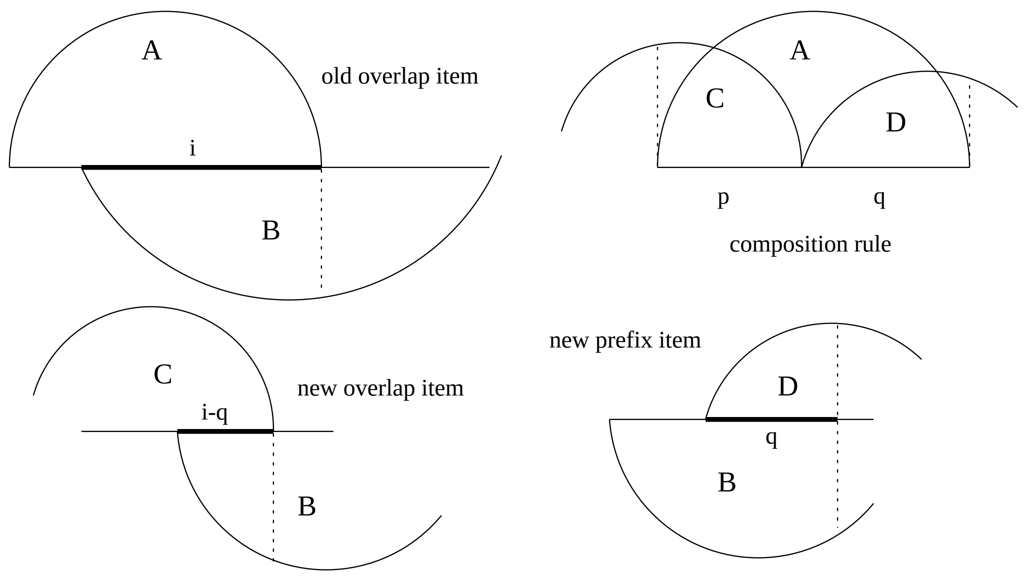

Figure 1: Splitting an overlap item, Case 2.

Observation 1 *Assume we have already checked that $|X| = |Y| = k$. Then the systems $\mathcal{S}_1$, $\mathcal{S}_2$ are equivalent iff the item $\alpha_0 = OV(X, Y, k)$ is valid (iff $val(X) = val(Y)$).*

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

```
*Initially*: $\Gamma = \{\alpha_0\}$
**invariant1**: $\Gamma \equiv \{\alpha_0\}$.
**invariant2**: $|\Gamma|$ is polynomial.
*Finally*: $\Gamma$ 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 item and possibly inserts other overlap items (related to the same variables as removed items).
Assume that for each atomic item α $SPLIT(\alpha) = \{\alpha\}$.

```
ALGORITHM EQUALITY_TEST ;
$\Gamma$ := { $\alpha_0$ };
while $\Gamma$ 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 $\Gamma$ are valid then return true
   else return false
```

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 linear size due to Lemma 1.3. The overlaps correspond to periods and due to Lemma 1.3 we can choose only a linear number of items of a given type $OV(A, B, *)$. We have quadratic number of pairs of variables, hence the upper bound is cubic. The operation satisfies:

$$(*) \qquad \begin{array}{c} \Delta \equiv Compact(\Delta) \text{ and} \\ |Compact(\Delta)| \leq c \cdot (n_1 + n_2)^3, \end{array}$$

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

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

Lemma 2.2 *The worst-case performance of the algorithm Equality_Test is polynomial with respect to the size of input.*

Proof. Let $n_1 = |\mathcal{S}_1|$, $n_2 = |\mathcal{S}_2|$.

Claim 1. There are at most $n_1 + n_2$ iterations of the algorithm.

If an item is split then one of its variables is replaced by "earlier" variables in the corresponding system. The variables in a given system are linearly ordered and we can "go back" at most $n_1 + n_2$ times. One can also imagine that we traverse a large tree which starts at α_0 and in which each branching corresponds to an application of $SPLIT$. We process this tree *top-down* level by level. The number of levels is linear. At each level there are possibly exponentially many items but (due to *Compact*) we process only polynomially many items.

Claim 2. $|\Gamma| \leq c \cdot (n_1 + n_2)^4$.

Let $\Gamma_i = \Delta_1(i) \cup \Delta_2(i)$ be the set of items after the ith iteration, where $\Delta_1(i)$ is the set of overlap items in Γ_i. Denote $k_i = |\Delta_1(i)|$ and $r_i = |\Delta_2(i)|$. Each item generates at most one non-overlap item. Hence:

$$r_{i+1} \leq k_i + r_i \text{ and } |\Gamma_i| = k_i + r_i.$$

Now the claim follows from the inequality (*) since k_i is the number of overlap items after an application of *Compact* and initially $r_0 = 0$.

Eventually, after some number of iterations all items in Γ are *atomic*. Then we apply the *validity test* for atomic items. This can be done in polynomial time, due to Fact 2.1. ∎

It follows directly from Lemma 2.2 that we can test equality in polynomial time. We find the first mismatch (in case of inequality) by a kind of a binary search in an exponentially long interval.

Theorem 2.3 *The Compressed Equality Testing can be done in polynomial time.*

3 The Pattern-Matching Algorithm

According to Fact 1.2 the fully compressed pattern matching is reduced to the 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 $val(X)$.

Let us fix the pattern $P = val(\mathcal{P})$. Let x be a string of length K and j be any position in this string. Define $Prefs(j,x)$ to be the lengths of subwords of x that end at position j in x and that are prefixes of P. Similarly, denote by $Suffs(j,x)$ the lengths of subwords of x that begin at position j in x and that are suffixes of P. Formally: $Prefs(j,x) = \{1 \le i \le j \ : \ x[j-i+1..j]$ is a a prefix of P $\}$.
$Suffs(j,x) = \{j \le i \le K \ : \ x[j..j+i-1]$ is a suffix of P $\}$.

Observation 2 *Let p, s, t be strings, then p occurs in $s \cdot t$ iff p occurs in s or p occurs in t, or $Solution(|p|, Prefs(|s|,s), Suffs(1,t)) \neq 0$.*

Define the operations of the *prefix-extension* and *suffix-extension*. For a word x define
$PrefExt(S,x) = \{i + |x| \ : \ i \in S$ and $P[1..i]\cdot x$ is a prefix of P $\}$.
$SuffExt(S,x) = \{i + |x| \ : \ i \in S$ and $x \cdot P[M-i+1..M]$ is a suffix of P $\}$.

Assume that $X_1, X_2, \ldots, X_n$ is a sequence of variables that appear in consecutive rules of the composition system $\mathcal{T}$ defining the text T. Denote

$$\begin{aligned} SUFF[j,i] &= Suffs(j, val(X_i)), \\ PREF[j,i] &= Prefs(j, val(X_i)). \end{aligned}$$

Observe that these tables depend on the pattern P, however it is convenient to assume further that P is fixed. Let k be the first position in $PREF[j,i]$, then all the other positions in $PREF[j,i]$ are of the form $k + p'$, where p' is a period of $P[1..k]$. Hence Lemma 1.3 implies directly the following fact.

Lemma 3.1
The sets $SUFF[j,i]$ and $PREF[j,i]$ are succinct, for any $1 \le i \le n$, $1 \le j \le |val(X_i)|$.

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 $b > s$. 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 $2n$ positions are being distinguished. They are called **fingers** and are denoted by $FINGERS(X_k)$. They correspond to *informers* in [5]. The positions in $FINGERS(X_k)$ are those predecessors of end-positions of variables that are in the word $val(X_k)$. Clearly for a fixed k in the set $FINGERS(X_k)$ there are at most two predecessors of end-positions of one variable, so that there are at most $2n$ positions in the set $FINGERS(X_k)$.

Lemma 3.2 *The sets $FINGERS(X_k)$ can be computed by a polynomial time algorithm.*

Proof. The algorithm finds the sets $FINGERS$ for consecutive variables starting from the variable X_n and ending with X_1. We have $FINGERS(X_n) = \{1, |val(X_n)|\}$. While considering the variable X_k the algorithm takes each finger b in $FINGERS(X_k)$ and puts $Pred[b, X_k]$ to appriopriate set $FINGERS$. ∎

In the second phase the pattern-matching algorithm inspects consecutive variables from X_1 to X_n in the composition system and for each finger b in the word $val(X_k)$ the sets $PREF[b,k]$ and $SUFF[b,k]$ are computed. We use this information to check whether there is an occurrence of the pattern inside $val(X_k)$.

Let S be a set of integers and d be an integer. Denote by $Cut(S,d)$ the subset of S consisting of numbers not greater than d.

Observation 3 *Assume S is given by its succinct representation. Then the succinct representation of the set $Cut(S,d)$ can be computed in polynomial time with respect to the number of bits in d and the size of the representation of S.*

Let S be a set of arithmetic progressions. Let $Compress(S)$ be the operation that glues pairs of progressions that can be represented by one arithmetic progression. Clearly the operation $Compress(S)$ can be implemented in polynomial time with respect to the number of progressions in S.

Below we describe the second phase of the algorithm. Note that whenever the algorithm needs values of the sets *SUFF* or *PREF* then they have been computed since they refer to fingers in previously considered variables.

Our next lemma says that the operations *PrefExt* and *SuffExt* can be implemented in polynomial time. Consider only the first of them, the second one is symmetric. We consider a set S which consists of one linear set. If there are polynomially many linear set-components of S, we deal with each of them separately.

Lemma 3.3 *Assume $\mathcal{W}$ is a composition system and $S = \{t_0, t_1, \ldots, t_s\} \subseteq [1 \ldots k]$ is a linear set given by its succinct representation, where $t_0 = k$ and strings $x_i = P[1..t_i]$, $0 \le i \le s$, are suffixes of $P[1..k]$. Then the representation of PrefExt$(S, val(\mathcal{W})$ can be computed in polynomial time.*

Proof.
The proof is similar to the proof of Lemma 7 in [8]. Assume the sequence $t_0, t_1, \ldots, t_s$ is decreasing. Denote $p = t_0 - t_1$. Since S is linear $p = t_i - t_{i+1}$ for $1 \le i \le s$. Thus the number p is a period of all words x_i. We need to compute all possible continuations of x_i's in P which match $val(\mathcal{W})$. Denote $y_i = P[1..t_i + |val(\mathcal{W})|]$ and $Z = P[1..k]{\cdot}val(\mathcal{W})$. Our aim is to find all i's such that y_i is a suffix of Z, $(0 \le i \le s)$. We call such i's *good* indices. The first mismatch to the period p in a string x is the first position (if there is any) such that $x[mismatch] \neq x[mismatch - p]$. The first mismatch can be computed using an equality-test algorithm from Theorem 2.3.

```
ALGORITHM $SECOND\_PHASE$ ;
for $k = 1$ to $n$ do
    if the rule for $X_k$ is $X_k = a$ then
        if $P = a$ then report occurrence and STOP
        else compute $PREF[1,k]$, $SUFF[1,k]$
    else $\{X_k = X_{i[s]} \cdot X_j^{[t]}$ for $i, j < k\}$
        $pref := Cut(PREF[|val(X_i)|, i], s)$; $suff := Cut(SUFF[1,j], t)$;
        $pos := Solution(|P|, pref, suff)$;
        if $pos \neq 0$ then report an occurrence and STOP
        else for each finger $b$ in $FINGERS(X_k)$ do
                if $b > s$
                then
                    $U := PrefExt(pref, val(X_k)[s+1..b]) \cup PREF[Pred[b, X_k], j]$;
                    $V := Cut(SUFF[Pred[b, X_k], j], |val(X_k)| - b + 1)$;
                else
                    $U := Cut(PREF[Pred[b, X_k], i], b)$;
                    $V := SuffExt(suff, val(X_k)[b..s]) \cup SUFF[Pred[b, X_k], i]$;
                $PREF[b,k] := Compress(U)$; $SUFF[b,k] := Compress(V)$;
```

If there is no mismatch in Z then Z is periodic with a period p. Then i is good iff y_i is periodic with a period p. Since all y_i are prefixes of y_0 it is enough to find the mismatch m in y_0. Good indices are the indices of words shorter than m.

If there is a mismatch at position z in Z then there is at most one good index. Clearly $z \geq k$ since there is no mismatch in $P[1..k]$. The mismatch in the suffix of Z starting at s in Z is at position $k - s$. The only good index i is such that the mismatch in y_i is at the same position as in corresponding suffix of Z. For all words y_i mismatches are at the same position as the mismatch in y_0. This allows to find easily the index i. Then it should be tested whether the index is really good.

In this way we compute the set of good indices. Observe that it consists of a subset of consecutive indices from the set S. So the corresponding set (the required output) of integers $\{|y_i| \ : i \text{ is a good index }\}$ is linear. This completes the proof. ∎

If the sets $SUFF[b,j]$, $PREF[b,j]$ has been already computed by the algorithm, then each of them consists of a polynomial number of linear sets, for $j < i$. Hence we can compute the sets $PREF[b,i]$ and $SUFF[b,i]$ in polynomial time using polynomially many time the algorithm from Lemma 3.3 to each of these linear sets. In this way we have shown that the algorithm $SECOND\_PHASE$ works in polynomial time. This completes the proof of our main result (Theorem 1.1).

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

Theorem 3.4 *Assume $\mathcal{S}$ is a composition system with n variables. Then we can compute in polynomial time a polynomial size representation of set Periods($val(\mathcal{S})$). The representation consists of a linear 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 easily all periods. ▮

4 Open Problem

Our method yields the first polynomial time algorithm for the **LZ *Fully Compressed Matching Problem*.** An interesting open problem remains on improving running time and storage requirements of an algorithm.

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] M. Crochemore and W. Rytter, *Text Algorithms,* Oxford University Press, New York (1994).

[5] M. Farach and M. Thorup, *String matching in Lempel-Ziv compressed strings,* to appear in STOC'95.

[6] M.R. Garey and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman (1979).

[7] R.M. Karp and M. Rabin, *Efficient randomized pattern matching algorithms,* IBM Journal of Research and Dev. 31, pp.249–260 (1987).

[8] M. Karpinski, W. Rytter and A. Shinohara, *Pattern-matching for strings with short description,* to appear in *Combinatorial Pattern Matching*, 1995

[9] D. Knuth, *The Art of Computing, Vol. II: Seminumerical Algorithms. Second edition.* Addison-Wesley (1981).

[10] A. Lempel and J.Ziv, *On the complexity of finite sequences, IEEE Trans. on Inf. Theory* 22, 75-81 (1976)

[11] M. Lothaire, *Combinatorics on Words.* Addison-Wesley (1993).

[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* 17, 8-19, 1984