Randomized Efficient Algorithms for Compressed Strings: the Finger-Print Approach

(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. Denote by $LZ(w)$ the coded form of a string w produced by **Lempel-Ziv encoding** algorithm. We consider several classical algorithmic problems for texts in the *compressed setting.* The first of them is the **equality-testing**: given $LZ(w)$ and integers i, j, k test the equality: $w[i \ldots i+k] = w[j \ldots j+k]$. We give a simple and efficient randomized algorithm for this problem using the *finger-printing* idea. The equality testing is reduced to the equivalence of certain *context-free grammars* generating single strings. The *equality-testing* is the *bottleneck* in other algorithms for compressed texts. We relate the time complexity of several classical problems for texts to the complexity $Eq(n)$ of *equality-testing.* Assume $n = |LZ(T)|$, $m = |LZ(P)|$ and $U = |T|$. Then we can compute the **compressed representations** of the sets of **occurrences** of P in T, **periods** of T, **palindromes** of T, and **squares** of T respectively in *times* $O(n \log^2 U \cdot Eq(m) + n^2 \log U)$, $O(n \log^2 U \cdot Eq(n) + n^2 \log U)$, $O(n \log^2 U \cdot Eq(n) + n^2 \log U)$ and $O(n^2 \log^3 U \cdot Eq(n) + n^3 \log^2 U)$, where $Eq(n) = O(n \log \log n)$. The randomization improves considerably upon the known deterministic algorithms ([7] and [8]).

1 Introduction

In the algorithmics of textual problems only recently the problems related to compressed objects were investigated ([1], [2], [3] and [8]). A very natural way

* 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`

and practical method of the text compression is the *LZ-compression* (see [12]). In this paper we consider several problems for LZ-compressed strings: *pattern matching* and computation of all **periods**, **palindromes** and **squares** of a given compressed string (without decompression). The first considered problem is the **Fully Compressed Matching Problem**:

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

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

The size of the problem is $n+m$, where $n = |LZ(T)|$ and $m = |LZ(P)|$. Denote $U = |T|$. Assume for simplicity that $m \leq n$, and $|P| \leq |T|$ then n determines the size of the compressed problem and U corresponds to the total size of the uncompressed problem. Note that in general U and the size of the set $\mathcal{S} = Occ(P,T)$ of all occurrences of P in T can be exponential with respect to n. Thus the algorithm which decompresses the pattern P and the text T could work in exponential time. Moreover by *computing* $\mathcal{S}$ we mean constructing its **compressed representation**, i.e. a data structure which size $|\mathcal{S}|$ is known and the first element of $\mathcal{S}$ is given explicitly. In case of pattern matching membership queries in the set $\mathcal{S}$ can be answered in time $O(n)$, and in case of palindromes and squares in time $O(n \log\log U)$.

The key concepts in our algorithms are *finger-printing*, *periodicity* and *linearly-succinct* representations of exponentially many periods.
Due to space limitations we omit several technical proofs.

2 The Lempel-Ziv compression and LZ-factorization.

We consider the same version of the LZ compression algorithm as one used in [5] (where it is called LZ1). Intuitively, LZ algorithm compresses the input word 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 $f_i = w[p \ldots q]$ and $q \leq |c_1 f_1 c_2 \ldots f_{i-1} c_i|$. 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:

$$c_1\ f_1\ c_2\ f_2\ c_3\ f_3\ c_4\ f_4\ c_5 = a\ a\ b\ ab\ b\ abb\ a\ ababbabba\ \#.$$

After identifying each subword f_i with its corresponding interval we obtain the following LZ encoding of the string:

$$LZ(aababbabbababbabb\#) = a[1,1]b[1,2]b[4,6]a[2,10]\#.$$

First we make the following modification of the LZ encoding, which allows to move all terminal symbols to the beginning of the code. The encoding starts from all terminal symbols from the word. The symbols are assumed to stay at positions $-1, -2\ldots$ of the word. Then each reference to a terminal symbol is replaced by an interval inside those positions. After this modification the LZ-code of the word in the Example is

$$ab[-2,-2][1,1][-1,-1][1,2][-1,-1][4,6][-2,-2][2,10]\#.$$

The Lempel-Ziv code defines a natural factorization of the encoded word into subwords which correspond to intervals in the code. The subwords are called **factors**. We deal later (to the end of the paper) with LZ-**factorization**:

$$w = f_1 f_2 f_3 \ldots f_k.$$

The LZ-factorization of w is of size $k = |LZ(w)|$. The last positions of factors are called **active points** and are denoted by a_i, hence $a_i = \sum_{j=1}^{i} |f_j|$ and $f_i = w[a_{i-1}+1\ldots a_i]$. We assume that together with the LZ-factorization we have the sequence of intervals $[l_i \ldots r_i]$ such that $r_i \le a_{i-1}$ and $f_i = w[l_i \ldots r_i]$, if $a_i > 0$.

In other words for each nontrivial factor f_i we know its occurrence $[l_i \ldots r_i]$ in the text *preceding* this factor.

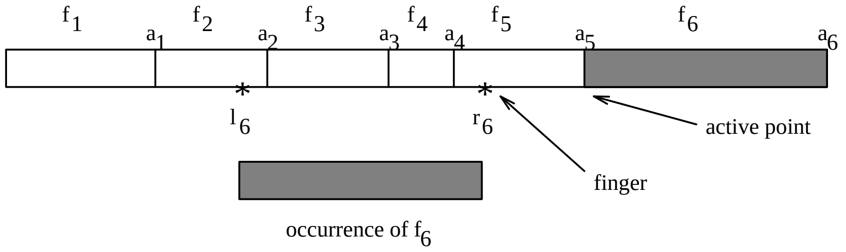

Fig. 1. A LZ-factorization of some string, the symbols '*' in the string correspond to (some) *fingers*. For example, r_6 is a *finger* since $r_6 = Pred(a_6)$.

Unfortunately the consideration of active points only is not sufficient. We have to introduce additional points, called *fingers*, which appear in the previous occurrences of factors f_i (see Figure 1).

Assume that the position b in the word w is in the factor f_i. Denote by $Pred[b]$ the position $b + r_i - a_i$ in w. The function $Pred$ (predecessor) defines a partial order "to be a predecessor" on positions in w. We define the set of **fingers** as a minimal set $\mathcal{F}$ of positions such that both ends of each factor are fingers and the predecessor of any finger is a finger. The number of all fingers is $O(|LZ(w)|^2)$ since for each position in w there is at most one predecessor of this position in each factor of w. This means also that in each factor there are

$O(|LZ(w)|)$ fingers. The first phase of our algorithms is a preprocessing which essentially consists of computing all fingers.

Lemma 1. *We can compute the set $\mathcal{F}$ of all fingers in $O(|LZ(w)|^2)$ time.*

Assume later that the set $\mathcal{F}$ of all fingers is precomputed.

3 Compressed representation of sets and general structure of their computation

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 $[1 \ldots U]$ 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 [8].

Lemma 2 LINEARLY-SUCCINCT SETS LEMMA.
The set $Periods(w)$ is linearly-succinct.

The lemma below shows that there are strings w_i which are "well compressible" but the sets of occurrences of a given pattern P in w_i are not well representable as families of linear sets.

Lemma 3. *There is a sequence of words w_i such that $|LZ(w_i)| = O(\log(|w_i|))$, and the sets of occurrences of a single letter in w_i are not representable as a union of polynomially many (with respect to $|LZ(w_i)|$) linear sets.*

Proof. Consider the sequence of words $\{w_i\}_{i \geq 0}$ defined by the following recurrences:

$$w_0 = a \quad w_i = w_{i-1} b^{|w_{i-1}|} w_{i-1} \text{ for } i \geq 1.$$

Let S_i be the set of positions of occurrences of the letter a in the word w_i. Clearly, $|S_i| = 2^i$. It can be shown that there is no sequence in S_i of length 3 which forms an arithmetic progression. Thus each decomposition of the set S_i into arithmetic sequences contains at least $|S_i|/2 = 2^{i-1}$ sequences and the set S_i is not linearly-succinct. However all the words w_i are "well compressible" since $|LZ(w_i)| \leq 4i + 1$ and $|w_i| = 3^i$. □

Due to Lemma 3 the set of occurrences of a pattern even in a *highly compressed* text is not necessarily *linearly-succinct*, so we have to introduce *compressed* representations which have more general meaning (but each *linearly-succinct* set has such *compressed* representation). Essentially our compressed representations consist of $O(n)$ linear sets, for example the representation for the set $Occ(P, T)$ of occurrences of the pattern P in T consists (*explicitly*) only of sets of occurrences which *overlap* active points. Despite the fact that the union of these sets is not necessarily the set $Occ(P, T)$ of all occurrences, it provides enough information about $Occ(P,T)$. By an *interval-occurrence* of a subword x we mean any interval $[i \dots j]$ such that $w[i \dots j] = x$. We say that a subword x **covers** a position r in w iff there is its *interval-occurrence* containing r. The data structures used in our algorithms consist of some sets of subwords of w which cover active points of w. The sets of subwords are represented by linearly-succinct sets and an additional constant-size information. Restriction to subwords which cover active points is justified by the following fact.

Fact 4 *If x is a subword of w then there is an occurrence of x in w which covers an active point.*

Our data structure $\mathcal{D}$ for the occurrences of a pattern, of palindromes or squares in a given string consists of a collection of sets $LeftSet(i)$, $RightSet(i)$ for each factor f_i. The sets represent certain subwords (related to patterns, palindromes or squares) which cover points $a_{i-1}+1$ and a_i, respectively. The sets can be represented in $O(\log U)$ space and their definitions differ according to the problem. The data structures allow to compute the number of all occurrences of the pattern, of palindromes or of squares in $O(n^2)$ time.

We start with a more exact description of the data structure for the pattern-matching problem. Denote by $\mathcal{D}_{pat}$ the representation of the set of occurrences of the pattern as a collection

$$\mathcal{D}_{pat} \;=\; \{\; \mathcal{D}_{pat}(i) \;:\; 1 \leq i \leq k \;\},$$

where $\mathcal{D}_{pat}(i)$ is the set of occurrences (starting positions) of the pattern *covering* the i-th active point and k is the number of factors. Hence in this case we have $LeftSet(i) = \emptyset$ and $RightSet(i) = \mathcal{D}_{pat}(i)$.

Lemma 5.
(1) *Each set $\mathcal{D}_{pat}(i)$ consists of a single arithmetic progression.*
(2) *Assume we know the set $\mathcal{D}_{pat}(t)$ for each factor f_t. Then for a given interval $[i \dots j]$ we can check all occurrences of the pattern within this interval in time $O(n)$.*

The data structure $\mathcal{D}_{pal}$ for palindromes consists of the sets of centers and radii of palindromes which cover active points. Then the set $Leftset(i)$, denoted by $LeftPal(i)$, contains all palindromes whose centers are inside f_i and which cover the left end of f_i, similarly $RightSet(i)$, denoted by $RightPal(i)$, corresponds to palindromes which cover the right end of f_i.

Lemma 6.
(1) *The sets $LeftPal(i)$ and $RightPal(i)$ can be represented in $O(\log U)$ space.*
(2) *Assume we know the data structure $\mathcal{D}_{pal}$. Then for a given interval $[i \dots j]$ we can test for the existence of a palindrome within this interval in time $O(n \log U)$.*

Similarly our data structure $\mathcal{D}_{square}$ for squares consists of the sets of centers of the words ww which cover active points. Then the set $LeftSet(i)$, denoted by $LeftSq(i)$, contains all centers of words ww which cover left end of f_i and are inside f_i. The set $RightSet(i)$, denoted by $RightSq(i)$, contains all centers of words ww which are inside f_i and cover the right end of f_i.

Lemma 7.
(1) *The sets $LeftSq(i)$ and $RightSq(i)$ can be represented in $O(\log U)$ space.*
(2) *Assume we know the data structure $\mathcal{D}_{square}$. Then for a given interval $[i \ldots j]$ we can test for the existence of a square within this interval in time $O(n \log U)$.*

Let $\mathcal{D}$ denote the compressed set $\mathcal{D}_{pat}$ or $\mathcal{D}_{pal}$. The computation of $\mathcal{D}$ consists of k iterations, where k is the total number of factors f_i. During the ith iteration the information related to *fingers* in the ith **active set** $ACTIVE(i)$ is computed. Set $ACTIVE(i)$ consists of all *fingers* which are in f_i or which are relevant with respect to further computation, this means they are predecessors of *fingers* in factors to the right of f_i, more formally:

$$ACTIVE(i) = \{\ b \in \mathcal{F}\ :\ b \in f_i \text{ or } RightMost(b) > i\},\ \text{ where}$$
$$RightMost(b) = \max\{j\ :(\ b = Pred(b') \text{ and } b' \in f_j \text{ and } b' \in \mathcal{F},\ j > i\ ) \text{ or } j = 0\}$$

Lemma 8. $|ACTIVE(i)| = O(n)$*, and all sets* $ACTIVE(i)$ *can be computed in* $O(n^2)$ *time.*

In the algorithms for each finger b we compute a certain set of subwords, denoted by $Info(b)$, which satisfies the following property: if b is not an endpoint of a factor, then all words in $Info(b)$ are inside the word $f_1 \ldots f_i$, where $i = RightMost(b)$.

The algorithms compute also the sets $Info(b, k)$ corresponding to all subwords in $Info(b)$ which are inside $f_1 \ldots f_k$. We compute the sets $Info(b, i)$ for consecutive is from 1 to n. Also $LeftSet(i)$ and $RightSet(i-1)$ are computed. Note that if a finger b is not active then the last computed set $Info(b, i)$ equals $Info(b)$.

```
ALGORITHM Scheme_of_Algorithms ;
  {computation of compressed representation $\mathcal{D}$};
  for $i := 1$ to $n$ do
      for each finger $b \in ACTIVE(i)$ do compute $Info(b, i)$
  for $i := 1$ to $n$ do
      compute $RightSet(i-1)$, $LeftSet(i)$
      on the basis of $Info(a_{i-1})$, $Info(a_{i-1}+1)$
```

4 The Randomized Equality-Test Algorithm

Our auxiliary problem is the **Compressed Equality Testing** problem:

Instance: a compressed text $LZ(w)$ and integers i, j, i', j'

Question: does $w[i..j] = w[i'..j']$? If “no” then find the first **mismatch**.

Assume for simplicity that the alphabet is binary, define the following one-to-one function *val* from the set of all strings to the set of 2 × 2 matrices as follows:

$$val(0) \ = \ \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \quad val(1) \ = \ \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$$

and

$$val(a_1 a_2 \ldots a_k) = val(a_1) \cdot val(a_2) \ldots \cdot val(a_k).$$

For a string $x = a_1 a_2 \dots a_k$ define

$$FingerPrint_p(x) = val(x) \ mod \ p \ ,$$

where $val(x)$ *mod* p is the matrix $val(x)$ with all components taken *modulo p*.

Lemma 9. *Let p be a random prime number from the interval* $[U \dots U^q]$ *and let* $w1$ *and* $w2$ *be two distinct strings of length at most* U*. Then*

$$Probability\{ \ FingerPrint_p(w1) = FingerPrint_p(w2) \ \} \leq 1/U^{q-1}$$

We say that the probability is *very small* iff it does not exceed $1/U^2$. We shall use twice the following simple property of *parse trees* in context-free grammars.

Lemma 10 CONCATENATION LEMMA.
Assume we have a context-free grammar G *in Chomsky's normal form and* $\mathcal{T}$ *is a parse tree for a string* w*. Then each subword of* w *can be generated from a concatenation of* $O(height(\mathcal{T}))$ *nonterminals.*

Theorem 11 RANDOMIZED EQUALITY TESTING.
Assume a string w *given in LZ-compressed form, then we can preprocess* w *in* $O(n^2 \log n)$ *time in such a way that each equality query about* w *can be answered in* $O(n \cdot \log\log n)$ *time with a very small probability of error.*

Proof. In [8, 11] the compressed strings were considered in terms of context-free grammars (grammars, in short) generating single words. We can prove:

Claim. Let $n = |LZ(w)|$. Then we can construct a context-free grammar G of size $O(n^2 \log n)$ which generates w and which is in the Chomsky normal form. Moreover the height of the derivation tree of w with respect to G is $O(n{\cdot}\log\log n)$.
Proof. (of the claim) The *fingers* partition the string w into consecutive subwords $sub_1, sub_2, \dots sub_p$, where $p = O(n^2)$. We associate a nonterminal A_i with each subword sub_i, this nonterminal generates sub_i in the constructed grammar. First we construct the grammar G', its *starting nonterminal* is a special nonterminal S with the production $S \rightarrow A_1 A_2 \dots A_p$. For each nonterminal A_i we have the production $A_i \rightarrow A_j A_{j+1} \dots A_r$, where $sub_i = sub_j sub_{j+1} \dots sub_r$. Such subwords exist due to properties of *fingers*. The constructed grammar G' has $O(n^2)$ nonterminals, but its total size can be cubic since the right sides of productions can be too long. We introduce $O(n^2)$ new nonterminals and build an *almost complete* regular binary tree $\mathcal{T}'$ generating $A_1 A_2 \dots A_p$. We have $height(\mathcal{T}') = O(\log n)$. Due to Lemma 10 we can replace each right side of a production in G' by a concatenation of $O(\log n)$ nonterminals. After that the right sides are of logarithmic size. And now the height of derivation trees is $O(n)$ since the productions correspond to operations *Pred*, and after linear number of applications of *Pred*s we are at the beginning of the string. However the grammar is not in Chomsky's normal form. We replace each right side by a special *small* derivation subtree of height $O(\log\log n)$. Then we obtain the grammar G. This completes the proof of the claim. □

Let G be the grammar from the claim, for each nonterminal A we compute $FingerPrint_p(A) = FingerPrint_p(w_A)$, where w_A is the string generated from A. The computation is done *bottom-up* using a kind of *dynamic programming*. Due to Lemma 10 each subword w' of w is generated from a concatenation $A_jA_{j+1}\dots A_r$ of at most $O(n\log\log n)$ nonterminals. The *finger-print* of w' can be computed as follows:

$$FingerPrint_p(w') =$$
$$= FingerPrint_p(A_j)\cdot FingerPrint_p(A_{j+1})\dots FingerPrint_p(A_r)$$

For each two subwords we can compute and compare their *finger-prints* in $O(n\log\log n)$ time. The probability of error is *very small* due to lemma 9. □

5 Computation of $\mathcal{D}_{pat}$, $\mathcal{D}_{pal}$ and $\mathcal{D}_{square}$

First we describe how to apply the *Compressed Equality Testing* in pattern matching algorithm. Let j be any position in the text T. Define $Pre(j,T)$ to be the lengths of subwords of T that end at position j in T and that are prefixes of the pattern P. Similarly, denote by $Suf(j,T)$ the lengths of subwords of T that begin at position j in T and that are suffixes of P. Formally:
$Pre(j,T) = \{1 \le i \le j\ :\ T[j-i+1\dots j]$ is a prefix of $P\}$.
$Suf(j,T) = \{1 \le i \le U-j+1\ :\ T[j\dots j+i-1]$ is a suffix of $P\}$.

Denote $Prefs(b,i) = Pre(b, f_1\dots f_i)$, $Suffs(b,i) = Suf(b, f_1\dots f_i)$. Let k be the biggest number in $Prefs(j,i)$, thus all other numbers in $Prefs(j,i)$ are of the form $k-p'$ where p' is a period of $P[1\dots k]$. Hence Lemma 2 implies directly the following fact.

Lemma 12. *The sets $Suffs(b,i)$ and $Prefs(b,i)$ are linearly-succinct.*

We are now able to give a sketch of the whole structure of the pattern matching algorithm. In the algorithm for each finger b we compute the sets $Prefs(b,i)$ and $Suffs(b,i)$, where $i = Rightmost[b]$. Hence in the case of pattern matching we have $Info(b,i) = Suffs(b,i) \cup Prefs(b,i)$. Using the information about $Prefs(b,i)$ and $Suffs(b,i)$ we can test for the pattern occurrence and compute number of occurrences by solving certain linear diofantine equations (see [9]) related to arithmetic progressions representing local occurrences.

Now we show how in the ith iteration the sets $Suffs(b,i)$ for active fingers b are computed. The sets $Prefs(b,i)$ are computed similarly. The set $Suffs(a_{i-1}+1,i)$ is properly computed on the basis of $Suffs(Pred[a_{i-1}+1], i-1)$. During the ith iteration, for each arithmetic sequence in $Suffs(a_{i-1}+1,i)$, we compute the set of **maximal pattern suffixes** which end at the positions of the sequence. As it is proved in [8] such a set can be represented in constant space and it is enough to compute such suffixes only for three positions in the sequence. Given a position i in w the length of the maximal pattern suffix which ends at i in w can be computed by **binary search**, using $\log|w|$ times the **equality test** procedure. Then the sets $Suffs(b,i)$ are computed for active fingers b. For each active finger (note that each finger inside f_i is active) in position $b > a_{i-1}$ we put to $Suffs(b,i)$, the suffixes from $Suffs(Pred[b], i-1)$ which are shorter than $a_i - b$, and for active fingers $b \le a_{i-1}$ the parts of sequences from $Suffs(a_{i-1}+1,i)$, for which the maximal suffixes extend over b. Similarly we compute the sets

Prefs(b, i). The time complexity is dominated by computing maximal prefixes and the sets *Prefs*(b, i), *Suffs*(b, i). Computing one maximal prefix is done in $O(Eq(m) \log U)$ time where $m = |LZ(P)|$. There are totally at most $2n \log U$ maximal prefixes and suffixes to compute. This gives the total time complexity $O((n \log^2 U) Eq(m))$ for computing maximal prefixes and suffixes. Given the maximal prefixes and suffixes all sets *Prefs*(b, i) and *Suffs*(b, i) can be computed in $O(n^2 \log U)$ time.

Theorem 13.
The compressed representation $\mathcal{D}_{pat}$ of the set of occurrences of a given pattern can be computed in $O((n \log^2 U) Eq(m) + n^2 \log U)$ time with a small probability of error.

As a side effect of our pattern-matching algorithm we can compute the set of all periods (use our string-matching algorithm with $P = T$).

Theorem 14. *Assume P is a compressed pattern. Then we can compute in $O((n \log U) Eq(m))$ time the compressed representation of the set Periods(P). The representation consists (in this case) of $\log U$ number of linear sets.*

Our algorithms for compressed **palindromes** and **squares** use ideas which apeared in [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 quite similar). Let $w[i \ldots j]$ 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 *Rad*$[1 \ldots |w|]$ be an array of radii of palindromes, this means that the entry *Rad*$[i]$ contains the maximal radius of a palindrome centered in i. The array *Rad* is a representation of all even length palindromes inside a word. Indeed, $w[i \ldots j]$ is an even palindrome iff $(i + j)$ is odd and *Rad*$[(i + j - 1)/2] > (j - i - 1)/2$. $\mathcal{D}_{pal}$ is a compressed representation of the array *Rad*.

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 $w[i + 1 \ldots |w|]$ (final palindromes in $w[1 \ldots i]$).

Lemma 15.
The sets $InitPal(w, i)$ and $FinPal(w, i)$ are linearly-succinct.

$\mathcal{D}_{pal}$ consists of sequences of numbers which are assigned to factors of the text T. With a factor f_i the sets $RightPal(i)$ and $LeftPal(i)$ are assigned. The sets are represented by arithmetic sequences of positions of centers of palindromes and each arithmetic sequence is equipped with the values of *Rad* for the first, last and a certain other element of the sequence. By Lemma 15 sets $RightPal(i)$ and $LeftPal(i)$ can be represented in $O(\log U)$ space. The basic component in the algorithm is the *equality-testing*.

Theorem 16. *The compressed representation $\mathcal{D}_{pal}$ of all palindromes in the compressed text can be computed in $O((n\log^2 U)Eq(n) + n^2 \log U)$ time with a small probability of error.*

An analogous theorem for squares looks as follows.

Theorem 17. *The compressed representation $\mathcal{D}_{square}$ of the set of all squares in the compressed text can be computed in $O((n^2\log^3 U)Eq(n)+n^3\log^2 U))$ time with small probability of error.*

Proof. (sketch) We concentrate on the set of squares in $LeftSq(i)$ for a factor f_i. The set $RightSq(i)$ can be considered similarly. Consider a square s which is in $LeftSq(i)$, its larger part is in $B = T[1\dots a_{i-1}]$, and its size is between 2^k and 2^{k+1}. The squares whose larger part is in $C = T[a_{i-1}+1\dots U]$ can be processed in the same way.

Similarly as in [4] we consider the *sample* v of size 2^{k-1} which is a suffix of B. Then $s = uvw$, for some u and w. An occurrence of the sample v is found in C by applying the *pattern-matching algorithm.* The size of s is the distance between two occurrences. The parts u, w are found by finding the first mismatch with respect to certain parts of the text, we omit the details. The **equality-test** is applied. In this way we find a single occurrence of a square in $LeftSq(i)$. □

6 Concluding remarks

We have shown that *randomization* helps in sequential computations for compressed texts. We are unable to construct deterministic NC algorithms for the main problems related to compressed texts. Does *randomization* helps in parallel computations too? We pose the following two **open questions.** Does there exist an RNC-algorithm for the *fully compressed pattern matching*? Are the following three problems $\mathcal{P}$*-complete*: *compressed equality*, *compressed palindrome* and *compressed square-free* testing?

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*, 296-307
5. M. Farach and M. Thorup, *String matching in Lempel-Ziv compressed strings,* in STOC'95, pp. 703-712.
6. R.M. Karp and M. Rabin, *Efficient randomized pattern matching algorithms,* IBM Journal of Research and Dev. 31, pp.249–260 (1987).
7. M. Karpinski, W. Plandowski and W. Rytter, The fully compressed string matching for Lempel-Ziv encoding. Technical Report, Institute of Informatics, Bonn University (1995)

8. M. Karpinski, W. Rytter and A. Shinohara, *Pattern-matching for strings with short description,* 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. W. Plandowski, *Testing equivalence of morphisms on context-free languages,* ESA'94, Lecture Notes in Computer Science 855, Springer-Verlag, 460–470 (1994).
12. J.Ziv and A.Lempel, *A universal algorithm for sequential data compression, IEEE Trans. on Inf. Theory* 17, 8-19, 1984

This article was processed using the LaTeX macro package with LLNCS style