Algorithms on compressed strings and arrays

Wojciech Rytter[1 2] *

[1] Department of Computer Science,Liverpool University, Chadwick Building, Peach Street, Liverpool L69 7ZF, U.K.

[2] Instytut Informatyki, Uniwersytet Warszawski, Banacha 2, 02–097 Warszawa, Poland.

Abstract. We survey the complexity issues related to several algorithmic problems for *compressed* one- and two-dimensional texts *without explicit decompression*: pattern-matching, equality-testing, computation of regularities, subsegment extraction, language membership, and solvability of word equations. Our basic problem is one- and two-dimensional pattern-matching together with its variations. For some types of compression the pattern-matching problems are infeasible (NP-hard), for other types they are solvable in polynomial time and we discuss how to reduce the degree of corresponding polynomials.

1 Introduction

In the last decade a new stream of research related to data compression has emerged: *algorithms on compressed objects*. It has been caused by the increase in the volume of data and the need to store and transmit masses of information in *compressed* form. The compressed information has to be quickly accessed and processed without explicit decompression. In this paper we consider several problems for compressed strings and arrays. The complexity of basic string problems in compressed setting for one dimensional texts is polynomial, but it jumps if we pass over to two dimensions. Our basic computational problem is the *fully compressed matching*:

Instance: $\mathcal{P} = \mathit{Compress}(P)$ and $\mathcal{T} = \mathit{Compress}(T)$, representing the compressed *pattern* and the compressed *text*.
Question: does $\mathit{Decompress}(\mathcal{P})$ occur in $\mathit{Decompress}(\mathcal{T})$?

We can change the way we formulate a problem instance by representing the pattern directly in uncompressed form , *i.e.*, as a text or an array P. This defines the *compressed matching* problem. We may also add to the problem instance the coordinates of a location of P within the text, and ask whether the text contains an occurrence of the pattern at this location. By representing the pattern in the compressed and uncompressed form we define the problems of *fully compressed pattern checking* and of *compressed pattern checking*.

* Supported by the grant KBN 8T11C03915

In this paper we are mostly interested in *highly compressed* objects, which means that the real size (of uncompressed object) is potentially exponential with respect to the size of its compressed representation. For one theoretical type of compression (in terms of morphisms) this can be even doubly exponential. In case of high compression the existence of polynomial time algorithms for many basic questions is a nontrivial challenge. High compression could be present in practical situations especially in compressing images. The theoretically interesting highly compressible strings are Fibonacci words. The n-th Fibonacci word is described by the recurrences: $F_1 := b;\ F_2 := a;\ F_n := F_{n-1} \cdot F_{n-2}$

For example

$F_{10} = abaababaabaababaababaabaababaabaababaababaabaababaababa$.

Two interesting examples of highly compressed arrays are the k-th rank square corner of Sierpinski triangle, denoted by S_k, see Figure 4, and the k-th order Hilbert array H_k, see Figure 3. S_k is a $2^k \times 2^k$ black and white array defined recursively: S_0 consists of a single black element, and S_k is a composition of 3 disjoint copies of S_{k-1} and totally white (blank) subarray, see Figure 1. The array H_k represents the sequence of moves in a *strongly recursive traversal* of $2^k \times 2^k$ array starting and finishing at fixed corners. The traversal of a square array is *strongly recursive* iff when entering any of its four quadrants all fields of this quadrant are visited (each one exactly once) before leaving the quadrant, and the same property holds for each quadrant and its sub-quadrants. The compressed matching problem for these particular examples is: check if an explicitly given word (array) P of total size m is a subword (subarray) of F_n ($H_n,\ S_n$), find the number of occurrences. Observe that in both cases we cannot simply construct F_n, H_n or S_n since their uncompressed sizes are exponential with respect to n. Such description of $F_k,\ S_k$ and H_h corresponds to one- and two-dimensional *straight-line programs* (SLP's), defined formally later . An SLP is a way of describing larger objects in terms of their (smaller) parts.

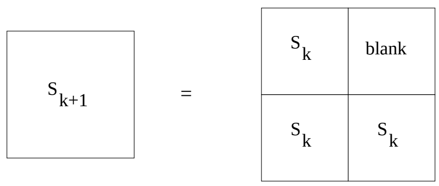

Fig. 1. The recursive structure of S_k.

2 Sequential searching in compressed texts

We discuss several types of 1-dimensional compression: run-length, straight-line programs (SLP's), LZ, LZW, and antidictionary compressions. We concentrate on LZ and SLP-compressions as the most interesting and closely related potentially exponential compressions. The key concepts in highly compressed matching algorithms are *periodicity* and *linearly-succinct* representations of exponentially many periods.

2.1 Periodicities in strings

A nonnegative integer p is a *period* of a nonempty string w iff $w[i] = w[i-p]$, whenever both sides are defined.

Lemma 1. [11]
If w has two periods p, q such that $p + q \leq |w|$ then $\gcd(p, q)$ is a period of w, where $\gcd$ means "greatest common divisor".

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. For example $Periods(aba) = \{0, 2, 3\}$.

Lemma 2. [34]
The set $Periods(w)$ is linearly-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. The size of the description, is the total number of bits in i, p, k. Denote by $Solutions(p, U, W) = (k, s)$ where k is any position $i \in U$ such that $i + j = p$ for some $j \in W$ and s is the number of such numbers k. The following lemma is used to count number of pattern occurrences in highly compressed texts.

Lemma 3.
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 $Solutions(c, U, W)$ in polynomial time with respect to $\log(N)$.

2.2 Run-length compression

The run-length compression (denoted by RLC(w)) of the string w is its representation in the form $w = a_1^{r_1} a_2^{r_2} \ldots a_k^{r_k}$, where a_i's are single symbols and $a_i \neq a_{i+1}$ for $1 \leq i < k$. Denote the size of the compressed representation by $n = |RLC(w)| = k$. We ignore here the sizes of integers and assume that each arithmetic operation takes a unit time.

Theorem 1. [3]
Assume we are given run-length encodings of the text T and the pattern P of sizes $n = |RLC(T)|$ and $m = |RLC(P)|$. Then we can check for an occurrence of P in T in $O(n + m)$ time.

Proof. Assume that the pattern and the text are nontrivial words: each of them contains at least two distinct letters. Let $T = a_1^{r_1} a_2^{r_2} \dots a_k^{r_k}$ and $P = b_1^{t_1} b_2^{t_2} \dots b_s^{t_s}$. Construct

$$T' = r_2 r_3 \dots r_{k-1},\ P' = t_2 t_3 \dots t_{s-1},$$
$$\alpha(T) = a_2 a_3 \dots a_{k-1} \text{ and } \alpha(P) = b_2 b_3 \dots b_{s-1}.$$

We search for all occurrences of P' in T' and simultaneously $\alpha(P)$ in $\alpha(T)$. For each starting occurrence i a constant-time additional work suffices to check if P occurs at i in T.

2.3 1-dimensional straight-line programs

A *straight-line program* (SLP) $\mathcal{R}$ is a sequence of assignment statements:

$$X_1 := expr_1;\ X_2 := expr_2; \dots;\ X_n := expr_n$$

where X_i are variables and $expr_i$ are expressions of the form:
$expr_i$ is a symbol of a given alphabet Σ, or $expr_i = X_j \cdot X_k$, for some $j, k < i$, where $\cdot$ denotes the concatenation of X_i and X_j.

Theorem 2. [34, 23]
The first occurrence and the number of all occurrences of a SLP compressed pattern in an SLP compressed text can be computed in polynomial time.

Proof. (Sketch)
For two variables X, Y define $Overlaps(X, Y)$ as the set of all positions i in Y such that the suffix of Y which starts at i is a prefix of X. De to Lemma 2 the sets $Overlaps(X, Y)$ are linearly-succinct. In the pattern matching problem we compute $Overlaps(X_i, P)$ and $Overlaps(P, X_j)$ for every variable X_i, X_j in the SLP describing T (bottom-up). Then we use Lemma 3 to check if there is an occurrence of the pattern overlapping the *splitting point*, see Figure 2, of some variable X_k.

Example The 5-th Fibonacci word is described by the following SLP:

$$X_1 := b;\ X_2 := a;\ X_3 := X_2 X_1;\ X_4 := X_3 X_2;\ X_5 := X_4 X_3$$

Using our algorithm it can be effectively found, for example, an occurrence (if there is any) of the Fibonacci word F_{220} in the Thue-Morse word ψ_{200} (see [40] and the last section for definition), despite the fact that *real* lengths of these strings are astronomic: $|\psi_{200}| = 2^{200}$ and $|F_{220}| \geq 2^{120}$.

Let $occ(P, T)$ be a word of length $|T|$ over the alphabet $\{0, 1\}$ such that $occ[i] = 1$ iff i is an ending position of an occurrence of P in T.

Theorem 3. *There is an SLP for $occ(P, T)$ which is of polynomial size with respect to $|SLP(T)|$ and $|P|$.*

Surprisingly the set of occurrences of a fixed pattern in a compressed text can have a very short SLP representation but its representation in terms of union of

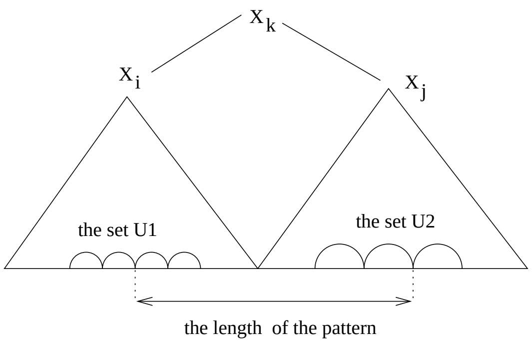

Fig. 2. A pattern occurs on a *splitting point* of the variable X_k iff $Solutions(|P|, U1, U2) \neq \emptyset$, where $U1 = Overlaps(P, X_i)$, $U2 = Overlaps(X_j, P)$.

arithmetic progressions is exponential. We take the sequence of words w_i over alphabet $\{a, b\}$ such that the k-th letter of w_i is a iff the number k, written in ternary, does not contain the digit 1. (The positions in w_i are counted starting from 0.)

Theorem 4. *There are strings w_i whose SLP representation is linear but the set of all occurrences of a single letter a in w_i is not linearly succinct.*

2.4 The Lempel-Ziv compression.

The LZ compression (see [60]) gives a very natural way of representing a string and it is a practically successful method of text compression. We consider the same version of the LZ algorithm as in [18] (this is called LZ1 in [18]). 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: $w = c_1 f_1 c_2 \dots f_k c_{k+1}$, 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 $f_i c_{i+1} \dots f_k c_{k+1}$ which appears in $c_1 f_1 c_2 \dots f_{i-1} c_i$. We can identify each f_i with an interval $[p, q]$, such that $f_i = w[p..q]$ and $q \leq |c_1 f_1 c_2 \dots f_{i-1} c_{i-1}|$. If we drop the assumption related to the last inequality then a *self-referencing* occurs (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 LZ encoding of the string. Hence

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

Lemma 4. *For each string w given in LZ-compressed form we can construct an SLP generating w of size $O(n^2)$, where $n = |LZ(w)|$.*

Theorem 5. [18] *The compressed pattern-matching for LZ-encoded texts can be done in $O(n \cdot \log^2(|T|/n) + |P|)$ time.*

When we pass to the fully compressed pattern-matching the situation is more complicated. We can use Lemma 4 to reduce the problem to the SLP-compressed matching. Another approach was used in [23], where a generalization of SLP's has been introduced and applied: a *composition system*. Denote by $Eq(n)$ the time to check subwords-equality of LZ-compressed texts. The technique of randomized *fingerprinting* has been used to show:

Theorem 6. [24]
Assume a string w is given in LZ-compressed form, then we can preprocess w in $O(n^2 \log n)$ time in such a way that each subword equality query about w can be answered in $O(n \cdot \log\log n)$ time with a very small probability of error.

Theorem 7. [23] *The Fully Compressed Matching Problem for LZ-compressed strings can be solved in $O(n \log n \log^2 |T| \cdot Eq(n \log n))$ time.*

Theorem 8. *The linearly-succinct representation of the set $Periods(\mathcal{P})$ can be computed in $O(n \log n \log^2 |T| \cdot Eq(n \log n))$ time.*

Theorem 9. [23] *We can test if an LZ compressed text contains a palindrome in $O(n \log n \log^2 |T| \cdot Eq(n \log n))$ time.*
We can test for square-freeness in $O(n^2 \log^2 n \log^3 |T| \cdot Eq(n \log n))$ time.

Theorem 10. [23]
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.

2.5 LZW-compression

The main difference between LZ and LZW-compression is in the choice of *code words*. Each next code word is of a form wa, where w is an earlier code-word and a is a letter. The text is scanned left to write, each time the next largest code-word is constructed. The code-words form a trie, and the text to be compressed is encoded as a sequence of names of prefixes of the trie. Denote by $LZW(w)$ the Lempel-Ziv-Welch compression of w. This type of compression cannot compress the string exponentially, it is less interesting from the theoretical point of view but much more interesting from the practical point of view.

Lemma 5. $|LZW(w)| = \Omega(\sqrt{|w|})$.

Theorem 11. [2] *The compressed matching problem for LZW-encoded strings can be done in $O(\min\{n + m^2,\ n \cdot \log(m) + m\})$ time.*

Theorem 12. [25]
The fully compressed matching problem for LZW-encoded strings can be done in $O((n + m) \cdot \log(n + m))$ time.

2.6 Texts compressed by using antidictionaries

The method of data compression that uses some "negative" information about the text is quite different from the other compression method. Assume in this subsection that the alphabet is binary. Let $AD(w)$ denote the set (called the *antidictionary*) of minimal *forbidden* factors of w, this means words which are not subwords of w. The minimality is in the sense of subword inclusion.

For example $AD(01001010) = \{000,\ 10101,\ 11\}$.

$AD(11010001) = \{0000,\ 111,\ 011,\ 0101,\ 1100\}$

The compression method processes w from left-to right symbol by symbol and produces the compressed representation $o_1 o_2 \dots o_N$, where each o_i is a single letter or an empty string. Assume that we scan the i-th letter a of w, if there is a word $ub \in AD(w)$ such that u is a suffix of $w[1,,i-1]$ then the i-th output word o_i is the empty word, otherwise it is a. In other words, if the i-th letter is predictable from $w[1..i-1]$ and $AD(w)$, we can skip the i-th letter, since it is *"redundant"*.

Denote $ADC(w) = (AD(w), o_1 o_2 \dots o_N, N)$, where $N = |w|$.
$ADC(w)$ is the *antidictionary compressed representation* of w.

Let $|AD(w)|$ denote the total length of all strings in $AD(w)$ and the size of compressed representation is defined as:

$$|ADC(w)| = |AD(w)| + |o_1 o_2 \dots o_N| + \log N.$$

Observe that N is an important part of the information necessary to identify w. For example

$$ADC(1^{1000}) = (\{0\}, \epsilon, 1000),\ ADC(1^{10}) = (\{0\}, \epsilon, 10).$$

This is a trivial example of an exponential compression. It can happen that $|ADC(w)| > |w|$, for example

$$ADC(11010001) = (\{0000,\ 111,\ 011,\ 0101,\ 1100\}, 110, 8)$$

Theorem 13. [10]
$ADC(w)$ can be computed in time $O(n+N)$, where $n = |ADC(w)|,\ N = |w|$.

Theorem 14. [57]
All occurrences of a pattern in w can be found in time $O(n + |P|^2 + r)$, where $n = |ADC(w)|$, and r is the number of occurrences of the pattern.

3 Sequential searching in compressed arrays

3.1 2-dimensional run-length encoding

For a 2-dimensional array T denote by $2RLC(T)$ the concatenation of run-length encodings of the rows of T.

Theorem 15. [4]
Assume we are given run-length encoding of a 2D-text T and an explicitly given 2D-pattern P, where $n = |2RLC(T)$ and $M = |(P)|$. Then we can check for an occurrence of P in T in $O(n+M)$ time.

3.2 2-dimensional straight-line programs

A 2D-text can be represented by a 2-dimensional straight-line program (SLP), that uses constants from the alphabet, and two types of assignment statements

Horizontal concatenation:
$A \leftarrow B \oslash C$, which concatenates 2D-texts B and C (both of equal height)

Vertical concatenation:
$A \leftarrow B \ominus C$, which puts the 2D-text B on top of C (both of equal width)

An SLP $\mathcal{P}$ of size n (we write $|\mathcal{P}| > n$) consists of n statements of the above form, where the result of the last statement is the compressed 2D-text. The result of a correct SLP, $\mathcal{P}$, is denoted *Decompress*$(\mathcal{P})$. We say that $\mathcal{P}$ is a *compressed form* of P. The *area* of $P =$ *Decompress*$(\mathcal{P})$, denoted $|P|$, can be exponential in $|\mathcal{P}|$.

Example. Hilbert's curve can be viewed as an image which is exponentially compressible in terms of SLP's. An SLP which describes the n^{th} Hilbert's curve, H_n, uses six (terminal) symbols , , , , , , and 12 variables $_i$, $_i$, $_i$, $_i$, $_i$, $_i$, $_i$, $_i$, $_i$, $_i$, $_i$, $_i$, for each $0 \leq i \leq n$. A variable with index i represents a text square of size $2^i \times 2^i$ containing part of a curve. The dots in the boxes show the places where the curve enters and leaves the box.

The 2D-text $T = {}_3$ describing the 3^{rd} Hilbert's curve is shown in Figure 3. It is composed of four smaller square 2D-texts $_2$, $_2$, $_2$ and $_2$ according to one of the composition rules. In the figure the black dots indicate how T was defined with statement $_3 \leftarrow ({}_2 \oslash {}_2) \ominus ({}_2 \oslash {}_2)$

The 1×1 text squares are described as follows.

$$\begin{array}{llll} _0 \leftarrow , & _0 \leftarrow , & _0 \leftarrow , & _0 \leftarrow , \\ _0 \leftarrow , & _0 \leftarrow , & _0 \leftarrow , & _0 \leftarrow , \\ _0 \leftarrow , & _0 \leftarrow , & _0 \leftarrow , & _0 \leftarrow , \end{array}$$

The text squares for variables indexed by $i \geq 1$ are rotations of text squares for the variables $_i$, $_i$, $_i$. These variables are composed according to the rules:

$$\begin{array}{l} _i \leftarrow ({}_{i-1} \oslash {}_{i-1}) \ominus ({}_{i-1} \oslash {}_{i-1}), \\ _i \leftarrow ({}_{i-1} \oslash {}_{i-1}) \ominus ({}_{i-1} \oslash {}_{i-1}), \\ _i \leftarrow ({}_{i-1} \oslash {}_{i-1}) \ominus ({}_{i-1} \oslash {}_{i-1}). \end{array}$$

Theorem 16. [9]
(1) *There exists a polynomial time randomized algorithm for testing equality of two 2D-texts, given their SLPs.*
(2) *Fully compressed pattern checking for 2D-texts is co-***NP***-complete.*
(3) Fully compressed matching for 2D-texts is $\Sigma_2^{\mathbf{P}}$-complete.
(4) *Compressed matching for 2D-texts is* **NP***-complete.*

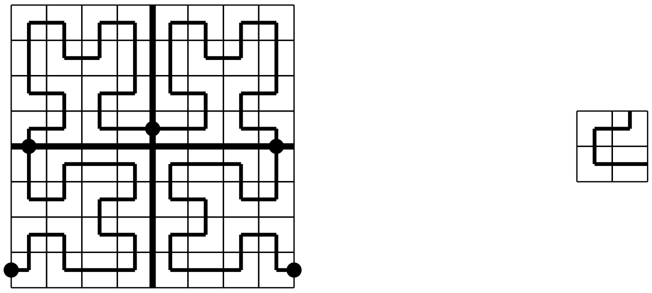

Fig. 3. An example of a 2D-text T and a pattern P. The pattern occurs twice in the text.

3.3 2D-compressions in terms of finite automata

Finite automata can describe quite complicated images, for example deterministic automata can describe the Hilbert's curve with a given resolution, see Figure 3, while weighted automata can describe even much more complicated curves, see also [13, 14, 16]. The automata can be also used as an effective tool to compress two-dimensional images, see [35, 15].

Our alphabet is $\Sigma = \{0,\ 1,\ 2,\ 3\}$, the elements of which correspond to four quadrants of a square array, listed in a fixed order. A word w of length k over Σ can be interpreted, in a natural way, as a unique address of a pixel x of a $2^k \times 2^k$ image (array), we write $address(x) = w$. The length k is called the *resolution* of the image. For a language $L \subseteq \Sigma^+$ denote by $Image_k(L)$ the $2^k \times 2^k$ black-and-white image such that the color of a given pixel x is black iff $address(x) \in L$. Formally, the weighted language is a function which associates with each word w a value $weight_L(w)$. A weighted language L over Σ and resolution k determine the gray-tone image $Image_k(L)$ such that the color of a given pixel x equals $weight_L(address(x))$. We define

$$Image_k(A) = Image_k(L(A)),$$

where $L(A)$ is the language accepted by A.

Example.
$Image(\Sigma^{k-1}(0 \cup 3))$ is the $2^k \times 2^k$ black-and-white chess-board.

Weighted finite automata and deterministic automata correspond to images of infinite resolution. Since we consider images of a given finite resolution k we can assume that the considered automata are acyclic.

Theorem 17. [20]
(1) *Ccompressed matching for deterministic automata can be solved in polynomial time.*
(2) *Compressed matching for weighted automata is NP-complete.*

3.4 2D-compression using LZ-encodings

A natural approach to (potentially exponential) compression of images is to scan a given two-dimensional array T in some specified order, obtain a linear version of T called *linear*(T), and then apply Lempel-Ziv encoding to the string *linear*(T).

The *Hilbert's curve* H_k corresponds to a *strongly regular* traversal of $2^k \times 2^k$ grid, starting in a bottom left corner of $n \times n$ array T, and ending at the right bottom corner, where $n = 2^k$. An example of H_3 is illustrated in Figure 3.

We denote now by *H-linear*(T) the linearization of T according to the Hilbert's curve. The *2LZ-compression* is defined as follows:

$$2LZ(T) \;=\; LZ(H\text{-}linear(T)).$$

Such type of encoding was already considered in [39]. $2LZ$-compression is as strong as *finite automata* compression, with respect to polynomial reduction.

Theorem 18.
If A describes an image T then $2LZ(T)$ is of polynomial size w.r.t. A.

Theorem 19.
Searching for an occurrence of a row of ones in a $2LZ$ compressed image is NP-hard.

Surprisingly there are black-and-white images whose $2LZ$ encoding is small and any *deterministic acyclic finite automata* encoding should be exponential.

Theorem 20.
For each m there is an image W_m such that $2LZ(W_m)$ is of size $O(m)$ but each deterministic automaton encoding the image W_m contains at least 4^{m-1} states.

4 Compressibility of subsegments

In this section we consider the problem of constructing a compressed representation of a part of a compressed object. The compressed representation of a part can be larger than that of the whole object. We start with finite-automata representations. In our considerations we may restrict to acyclic automata since we consider only finite resolution images.

Example. $Image(\{0,1,2\}^k) = S_k$ is the $2^k \times 2^k$ black-and-white square part of Sierpinski's triangle. see Figure 4 for the case $k = 4$. The corresponding smallest acyclic deterministic automaton accepting all paths describing black pixels has 5 essential states but there are needed 9 essential states to describe the 8×8 subarray R of S_4 indicated in Figure 4. (The state is *essential* iff it is on an accepting path, other states are treated as redundant.)

Theorem 21. [9]
Assume the compression is in terms of deterministic automata. Let n be the size

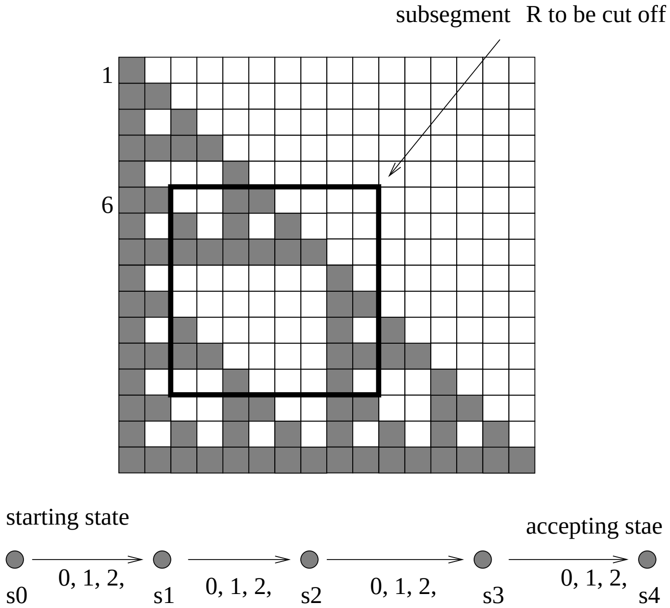

Fig. 4. The image S_4 and its smallest acyclic automaton. Edges which are not on an accepting path are disregarded.

of a deterministic automaton describing $\mathcal{T}$.
(a) *The compressed representation of a subsquare R of T can be computed in $O(n^{2.5})$ time.* **(b)** *For each subimage $\mathcal{R}$ of an image $\mathcal{T}$ there is a deterministic automaton describing $\mathcal{R}$ of size $O(n^{2.5})$. There are images $\mathcal{T}$ and their subimages $\mathcal{R}$ such that the smallest deterministic automaton for $\mathcal{R}$ requires $\Omega(n^{2.5})$ states.*

The situation is much different for 2-dimensional straight-line programs.

Theorem 22. [29]
For each n there exists an SLP *of size n describing a text image A_n and a subrectangle B_n of A_n such that the smallest* SLP *describing B_n has exponential size.*

5 Parallel searching in compressed texts and arrays

The difference between sequential and NC-computations for compressed texts can be well demonstrated by the following problem: compute the symbol $T[i]$ where T is the text given in its LZ-version, and i is given in binary.

This task has a trivial sequential linear time algorithm (which performs computations sequentially in the order the recurrences are written). However if we ask for an NC-algorithm for the same problem the situation is different, and it becomes quite difficult. Any straightforward attempt of using the doubling technique and squaring the matrix of positions fails, since the number of positions in the text defined by n recurrence equations can be $\Omega(2^n)$.

Theorem 23. [26]
(1) *The problem of testing for any occurrence of an uncompressed pattern in a LZ-compressed text is P-complete.*
(2) *The problem of computing a symbol on a given position in a LZ-compressed text is P-complete.*

Theorem 24. [26] *The SLP-compressed matching problem can be solved in* $O(\log(m) \cdot \log(n))$ *time with* $O(n)$ *processors.*

Theorem 25. [26]
The pattern-matching problem for SLP-compressed 2d-texts can be solved in
(1) $O(\log^2(n+m))$ *time with* $O(n^2 \cdot m + n^6)$ *processors, or*
(2) $O(n + \log m)$ *time with* $O(n \cdot (n+m))$ *processors.*

Theorem 26. [25]
There is an almost optimal NC algorithm for fully compressed LZW-matching.

Theorem 27. [6, 12]
The computation of $LZW(w)$ *is* P*-complete.*
The computation of $LZ(w)$ *is in* NC.

6 The compressed language membership problems

The language membership problem is to check if $w \in L$, given $Compress(w)$ and a description of a formal language L.

Theorem 28.
(a) *We can test the membership problem for* LZ*-compressed words in a language described by given regular expression* W *in* $O(n \cdot m^3)$ *time, where* $m = |W|$.
(b) *We can decide for* LZ*-compressed words the membership in a language described by given determiniatic automaton* M *in* $O(n \cdot m)$ *time, where* m *is the number of states of* M.

We use the following problem to show NP-hardness of several compressed recognition problems.

SUBSET SUM problem:
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 binary representation of numbers in A and K.
Question: Is there a subset $A' \subseteq A$ such that the sum of the elements in A' is exactly K?

Lemma 6. *The problem SUBSET SUM is NP-complete.*

Proof. see [31], and [22], pp. 223.

Theorem 29. [34, 53]
Testing the membership of a compressed unary word in a language described by a star-free *regular expression with compressed constants is NP-complete.*

It is not obvious if the previously considered problem is in NP for regular expressions containing the operation *, in this case there is no polynomial bound on the length of accepting paths of A. There is a simple argument in case of unary languages. In the proof of the next theorem an interesting application of the Euler path technique to a unary language recognition is shown.

Theorem 30. [53]
(a) *The problem of checking membership of a compressed unary word in a language described by a given regular expression with compressed constants is in NP.*
(b) *The problem of checking membership of a compressed word in a language described by a semi-extended regular expression is NP-hard.*

Theorem 31.
The problem of checking membership of a compressed word in a given linear context-free language L is NP-hard, even if L is given by a context-free grammar of a constant size.

Proof.
Take an instance of the subset-sum problem with the set $A = \{a_1, a_2, \dots, a_n\}$ of integers and an integer K. Define the following language:

$$L = \{d^R \$ d^{v_1} \# d^{v_2} \# \dots \# d^{v_t} \ : \ t \geq 1 \text{ and there is a subset } A' \subseteq \{v_1, \dots, v_t\} \text{ such that } \textstyle\sum_{u \in A'} u \ = \ R\} \ ?$$

L is obviously a linear context-free language generated by a linear context-free grammar of a constant size. We can reduce an instance of the subset sum-problem to the membership problem:

$$d^K \$ d^{a_1} \# d^{a_2} \# \dots \# d^{a_n} \ \in \ L.$$

Theorem 32.
(a) *The problem of checking membership of a compressed word in a given linear cfl is in $NSPACE(n)$.*
(b) *The problem of checking membership of a compressed word in a given cfl is in $DSPACE(n^2)$.*

Proof.
We can easily compute in linear space a symbol on a given position in a compressed input word. Now we can use a space-efficient algorithm for the recognition of context-free languages. It is known that linear languages can be recognized

in $O(logN)$ nondeterministic space and general cfls can be done in $O(log^2 N)$ deterministic space, where N is the size of the uncompressed input word. In our case $N = O(2^n)$, this gives required $NSPACE(n)$ and $DSPACE(n^2)$ complexities.

Theorem 33.
The problem of checking membership of a compressed unary word in a given cfl is NP-complete.

7 Word equations

Word equations are used to describe properties and relations of words, e.g. pattern-matching with variables, imprimitiveness, periodicity, and conjugation, see [30]. The main algorithm in this area was Makanin's algorithm for solving word equations, see [41]. The time complexity of the algorithm is too high, and the algorithm is too complicated. Recently much simpler algorithms were constructed by W. Plandowski [51, 52].

Let Σ be an alphabet of constants and Θ be an alphabet of variables. We assume that these alphabets are disjoint. A word equation E is a pair of words $(u, v) \in (\Sigma \cup \Theta)^* \times (\Sigma \cup \Theta)^*$ usually denoted by $u = v$. The *size* of an equation is the sum of lengths of u and v. A *solution* of a word equation $u = v$ is a morphism $h : (\Sigma \cup \Theta)^* \to \Sigma^*$ such that $h(a) = a$, for $a \in \Sigma$, and $h(u) = h(v)$. For example assume we have the equation

$$abx_1x_2x_2x_3x_3x_4x_4x_5 = x_1x_2x_3x_4x_5x_6,$$

and the length of x_i's are consecutive Fibonacci numbers. Then the solution $h(x_i)$ is the i-th Fibbonaci word.

It is known that the solvability problem for word equations is NP-hard, even if we consider (short) solutions with the length bounded by a linear function and the right side of the equation contains no variables, see [5].
The main open problem is to show the following:

Conjecture A: The problem of solving word equations is in NP.

Conjecture B: Let $\mathcal{N}$ be the minimal length of the solution (if one exists). Then $\mathcal{N}$ is singly exponential w.r.t. n.

The author believes that both questions have positive answers.

A motivation to consider compressed solutions follows from the following fact (which is an application of a fully compressed pattern-checking, in this case checking occurrence of one compressed string at the beginning of another one).

Lemma 7.
If we have LZ-encoded values of the variables then we can verify the word equation in polynomial time with respect to the size of the equation and the total size of given LZ-encodings.

Theorem 34.
Assume N is the size of minimal solution of a word equation of size n. Then each solution of size N can be LZ-compressed to a string of size $O(n^2 \log^2(N)(\log n + \log\log N))$.

As a direct consequence of Lemma 7 and Theorem 34 we have:

Theorem 35.
Conjecture B implies conjecture A.

Theorem 36.
Assume the length of all variables are given in binary by a function f. Then we can test solvability in deterministic polynomial time, and produce polynomial-size LZ-compression of the lexicographically first solution (if there is any).

8 Morphic representations.

There is a classical (in terms of formal language theory) short description of strings in terms of morphisms. Assume $w = \phi^k(a)$, where ϕ is a morphism and a is a letter in the alphabet. Words of the type $\phi^k(a)$ can be interpreted as instructions in a "turtle language" to draw fractal images, see [49]. Hence the calculation of the i-th letter has a practical meaning when computing a local structure of a fractal without computing the whole object. The pair (k, ϕ) can be treated as short morphic description of w, denoted by $morphic\_desc(w)$. The length $n = |morphic\_desc(w)|$ of the description is the size of the binary representation of k and ϕ.

For some morphisms the computation of the i-th letter could be especially simple. This is the case for the Thue-Morse morphism

$$\Psi(0) = 01,\ \Psi(1) = 10$$

Assume we count positions starting from 0. Let $bin(i)$ be the binary representation of i. Then:

$$\psi^k(0) = 1 \Leftrightarrow bin(i) \text{ contains an odd number of ones}$$

For example the 1024-th position of $\psi^{100}(0)$ is 1 since $bin(1024) = 10000000000$ contains odd number of ones. However such simple computation of the i-th letter does not apply to every morphism.

Theorem 37. *Let ϕ be a morphism. There is a polynomial time algorithm to compute the i-th letter of $\phi^k(a)$, where the input size is the total number of bits for i, ϕ, and k.*

Instead of morphic functions ϕ we can use a finite-state transducer function λ_A where A is a finite state transducer (deterministic finite automaton with an output function). The replacement of ϕ by λ_A has dramatic consequences.

Theorem 38. *[56]*
Let A be a finite state transducer. The problem of computing the i-th letter of $\lambda_A^k(a)$ is EXPTIME-hard.

9 Final remarks

We have surveyed complexity-theoretical results related to the processing of large compressed texts and arrays without decompression. However we have not discussed one important aspects: practicality of algorithms. Recently many related practical issues were investigated, see [32, 33, 45–47]. The whole area is in an initial stage.

References

1. A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The design and analysis of computer algorithms*, *Addison-Wesley*, Reading, Mass., 1974.
2. A. Amir, G. Benson and M. Farach, Let sleeping files lie: pattern-matching in Z-compressed files. *Journal of Computer and System Sciences*, 1996, Vol. 52, No. 2, pp. 299-307
3. A. Amir, G. Benson, Efficient two dimensional compressed matching, *Proceedings of the* 2^{nd} *IEEE Data Compression Conference*, pp. 279–288 (1992).
4. A. Amir, G. Benson and M. Farach, Optimal two-dimensional compressed matching, *Journal of Algorithms*, 24(2):354-379, August 1997
5. Angluin D., Finding patterns common to a set of strings, *J.C.S.S.*, 21(1), 46-62, 1980.
6. S. De Agostino, P-complete problems in data compression, *Theoretical Computer Science* 127, 181-186, 1994
7. S. De Agostino, Pattern-matching in text compressed with the ID heuristic, in J. Atorer, and M. Cohn (editors), *Data Compression Conference* 1998, IEEE Computer Society, pp. 113-118
8. M.F. Barnsley, L.P. Hurd, Fractal image compression, A.K.Peters Ltd. 1993
9. P. Berman, M. Karpinski, L. L. Larmore, W. Plandowski, and W. W. Rytter, On the Complexity of Pattern Matching for Highly Compressed Two-Dimensional Texts, *Proceedings of the* 8^{th} *Annual Symposium on Combinatorial Pattern Matching* LNCS 1264, Edited by A. Apostolico and J. Hein, (1997), pp. 40–51.
10. M. Crochemore, F. Mignosi, A. Restivo, S. Salemi, Text compression using antidictionaries, *ICALP* 1999
11. M. Crochemore and W. Rytter, *Text Algorithms,* Oxford University Press, New York (1994).
12. M. Crochemore, W. Rytter, Efficient parallel algorithms to test square-freeness and factorize strings, Information Processing Letters, 38 (1991) 57-60
13. K. Culik and J. Karhumaki, Finite automata computing real functions, *SIAM J. Comp* (1994).
14. K. Culik and J. Kari, Image compression using weighted finite automata, *Computer and Graphics* 17, 305-313 (1993).
15. K. Culik and J. Kari, *Fractal image compression: theory and applications*, (ed. Y. Fisher), Springer Verlag 243-258 (1995).
16. D. Derencourt, J. Karhumäki, M. Letteux and A. Terlutte, On continuous functions computed by real functions, *RAIRO Theor. Inform. Appl.* 28, 387-404 (1994).
17. S. Eilenberg, *Automata, Languages and Machines*, Vol.A, Academic Press, New York (1974).

18. M. Farach and M. Thorup, String matching in Lempel-Ziv compressed strings, *Proceedings of the* 27^{th} *Annual Symposium on the Theory of Computing* (1995), pp. 703–712.
19. M. Farach, S. Muthukrishnan, Optimal Parallel Dictionary Matching and Compression *SPAA* 1995
20. J.Karhumaki, W.Plandowski,W.Rytter, Pattern matching for images generated by finite automata, *FCT'97*, in LNCS Springer Verlag 1997
21. M. Gu, M. Farach, R. Beigel, An Efficient Algorithm for Dynamic Text Indexing (SODA '94)
22. M.R. Garey and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness,* Freeman, New York (1979).
23. L. Gąsieniec, M. Karpinski, W. Plandowski and W. Rytter, Efficient Algorithms for Lempel-Ziv Encoding, *Proceedings of the* 5^{th} *Scandinavian Workshop on Algorithm Theory*. Springer-Verlag (1996).
24. L.Gasieniec, M.Karpinksi, W.Plandowski, W. Rytter, Randomized algorithms for compressed texts: the finger-print approach, *Combinatorial Pattern Matching* 1996, Lecture Notes in Comp. Science, Springer Verlag 1996
25. L.Gasieniec, W. Rytter, Almost optimal fully compressed LZW-matching, in *Data Compression Conference*, IEEE Computer Society 1999
26. L.Gasieniec, A.Gibbons, W. Rytter, The parallel complexity of pattern-searching in highly compressed texts, in *MFCS* 1999
27. A. Gibbons, W. Rytter, Efficient parallel algorithms, Cambridge University Press 1988
28. O. H. Ibarra and S. Moran, Probabilistic algorithms for deciding equivalence of straight-line programs, *JACM* 30 (1983), pp. 217–228.
29. J. Karhumaki, W. Plandowski, W.Rytter, The compression of subsegments of compressed images, in *Combinatorial Pattern Matching* 1999
30. Karhumäki J., Mignosi F., Plandowski W., The expressibility of languages and relations by word equations, *in ICALP'97*, LNCS 1256, 98-109, 1997.
31. R.M. Karp, Reducibility among combinatorial problems, in *"Complexity of Computer Computations", Plemum Press, New York"*, 1972 (editors R.E. Miller and J.W. Thatcher)
32. T. Kida, M. Takeda, A. Shinohara, S. Arikawa, Sfift-and approach to pattern-matching in LZW compressed text, *Combinatorial Pattern-Matching* 1999, LNCS 1645, pp. 1-13
33. T. Kida, M. Takeda, A. Shinohara, M. Miyazaki, S. Arikawa, Multiple pattern-matching in LZW compressed text, in J. Atorer, and M. Cohn (editors), *Data Compression Conference* 1998, IEEE Computer Society, pp.103-112
34. M. Karpinski, W. Rytter and A. Shinohara, Pattern-matching for strings with short description, *Nordic Journal of Computing*, 4(2):172-186, 1997
35. J. Kari, P. Franti, Arithmetic coding of weighted finite automata, *RAIRO Theor. Inform. Appl.* 28 343-360 (1994)
36. R.M. Karp and M. Rabin, *Efficient randomized pattern matching algorithms,* IBM Journal of Research and Dev. 31, pp.249–260 (1987).
37. D. Knuth, *The Art of Computing, Vol. II: Seminumerical Algorithms. Second edition.* Addison-Wesley (1981).
38. A. Lempel and J. Ziv, On the complexity of finite sequences, *IEEE Trans. on Inf. Theory* 22 (1976) pp. 75–81.
39. A. Lempel and J. Ziv, Compression of two-dimensional images sequences, *Combinatorial algorithms on words* (ed. A. Apostolico, Z. Galil), Springer-Verlag (1985) pp. 141–156.

40. M. Lothaire, *Combinatorics on Words.* Addison-Wesley (1993)
41. Makanin, G.S., The problem of solvability of equations in a free semigroup, *Mat. Sb.*, Vol. 103,(145), 147-233, 1977. English transl. in *Math. U.S.S.R.* Sb. Vol 32, 1977.
42. U. Manber, A text compression scheme that allows fast searching directly in the compressed file, *ACM Transactions on Information Systems*, 15(2), pp. 124-136, 1997
43. M. Miyazaki, A. Shinohara, M. Takeda, An improved pattern-matching algorithm for strings in terms of straight-line programs, *Combinatorial Pattern-Matching* 1997, LNCS 1264, pp. 1-11
44. R. Motwani, P. Raghavan, *Randomized Algorithms*, Cambridge University Press (1995).
45. E. de Moura, G, Navarro, N. Ziviani, R. Baeza-Yates, Direct pattern-matching on compressed text, in SPIRE, pp. 90-95, IEEE CS Press, 1998
46. E. de Moura, G, Navarro, N. Ziviani, R. Baeza-Yates, Fast sequential searching on compressed texts allowing errors, *21st Annual Int. ACM SIGIR Conference on Research and Development in Information retrieval*, pp. 298-306, York Press 1998
47. G. Navarro, M. Raffinot, A general practical approach to pattern-matching over Zil-Lempel compressed text, *Combinatorial Pattern-Matching* 1999, LNCS 1645, pp. 14-36
48. Ch. H. Papadimitriou, *Computational Complexity*, Addison Wesley (1994).
49. H.O.Peitgen, P.H. Richter, The beauty of plants, Springer-Verlag 1986
50. W. Plandowski, Testing equivalence of morphisms on context-free languages, *Proceedings of the* 2^{nd} *Annual European Symposium on Algorithms* (ESA'94), LNCS 855, Springer-Verlag (1994), pp. 460–470.
51. W. Plandowski, Solvability of word equations with constants is in NEXPTIME, *STOC* 1999
52. W. Plandowski, Solvability of word equations with constants is in P-SPACE, *FOCS* 1999
53. W. Plandowski, W. Rytter, Complexity of compressed recognition of formal languages, in *"Jewels forever"*, Springer Verlag 1999 (ed. J. Karhumaki)
54. W.Plandowski, W.Rytter, Applying Lempel-Ziv encodings to the solution of word equations, *ICALP* 1998
55. J. Schwartz, Fast probabilistic algorithms for verification of polynomial identities, *J. ACM* 27 (1980) pp. 701–717.
56. J. Shallit, D. Swart, An efficient algorithm for computing the i-th letter of $\phi^n(a)$, *SODA* 1999
57. Y. Shibata, M. Takeda, A. Shinohara, S. Arikawa, Pattern-matching in text compressed by using antidictionaries, *Combinatorial Pattern-Matching* 1999, LNCS 1645, pp. 37
58. J. Storer, *Data compression: methods and theory,* Computer Science Press (1988).
59. R. E. Zippel, Probabilistic algorithms for sparse polynomials, *Proceedings of the International Symposium on Symbolic and Algebraic Manipulation* (EUROSAM '79) LNCS 72, Springer-Verlag (1979), pp. 216–226.
60. J. Ziv and A. Lempel, A Universal algorithm for sequential data compression, *IEEE Transactions on Information Theory* IT–23 (1977), pp. 337–343.