Pattern-Matching for Strings with Short Descriptions

Marek Karpinski * Wojciech Rytter † Ayumi Shinohara **

Abstract. We consider strings which are succinctly described. The description is in terms of straight-line programs in which the constants are symbols and the only operation is the concatenation. Such descriptions correspond to the systems of recurrences or to context-free grammars generating single words. The descriptive size of a string is the length n of a straight-line program (or size of a grammar) which defines this string. Usually the strings of descriptive size n are of exponential length. *Fibonacci* and *Thue-Morse words* are examples of such strings. We show that for a pattern P and text T of descriptive sizes m, n, an occurrence of P in T can be found (if there is any) in time polynomial with respect to n. This is nontrivial, since the actual lengths of P and T could be exponential, and none of the known string-matching algorithms is directly applicable. Our first tool is the periodicity lemma, which allows to represent some sets of exponentially many positions in terms of feasibly many arithmetic progressions. The second tool is arithmetics: a simple application of Euclid algorithm. Hence a textual problem for exponentially long strings is reduced here to simple arithmetics on integers with (only) linearly many bits. We present also an NP-complete version of the pattern-matching for shortly described strings.

* Deptartment of Computer Science, University of Bonn, 53117 Bonn, and International Computer Science Institute, Berkeley, California. Research partially supported by the DFG Grant KA 673/4-1, and by the ESPRIT BR Grants 7097 and ECUS 030. Email: marek@cs.uni-bonn.de

† Deptartment of Computer Science, University of Bonn, 53117 Bonn, and Institute of Informatics, Warsaw University, 02-097 Warszawa. Supported by the DFG Grant Bo 56/142-1. Email: rytter@mimuw.edu.pl

** Deptartment of Computer Science, University of Bonn, 53117 Bonn, and Research Institute of Fundamental Information Science, Kyushu University 33, Fukuoka 812, Japan. Email: ayumi@rifis.kyushu-u.ac.jp

1 Introduction

We describe algorithms for *implicit* pattern-matching problems for some well structured and exponentially long strings, given in the form of a succinct description. The *descriptive size* n of such strings is the size of their description, while their *real size* N is the actual length of the string, assuming it is *explicitly written*. The size of the whole problem is n. Usually $N = \Omega(2^{c\,n})$.

In our algorithms we cannot write such long strings *explicitly*. Fortunately, each position in such strings can be written with only linear number of bits. Hence the size of the output is small: the output of our algorithm is an occurrence of one long string in another very long string. The input consists of short descriptions of the strings in terms of *straight-line* programs. We strengthen (in a nontrivial way) a result of [7], where quite sophisticated polynomial-time algorithm for equality of two strings generated by grammars was given. The *equality-test* algorithm from [7] is not directly applicable here since there are exponentially many positions, where the equality between the pattern and the text can happen. However we use this algorithm as one of the basic subroutines.

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

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

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_j and X_k.

For each variable X_i, denote by $\nu(X_i)$ the value of X_i after the execution of the program. $\nu(X_i)$ is the string described by X_i. Denote by R the string described by (the value of) the program $\mathcal{R}$: $R = \nu(\mathcal{R}) = \nu(X_n)$. The size $|\mathcal{R}|$ of the program $\mathcal{R}$ is the number n, it is also called the *descriptive size* of the generated string $R = \nu(\mathcal{R})$.

R is called a *string with short description*, since usually $|R|$ is very long (exponentially) with respect to its descriptive size $n = |\mathcal{R}|$.

We say that $\mathcal{R}$ describes a *long string*. If we consider a string in a usual sense (the description is by giving the string *explicitly*) then we call such string an *explicit string*.

For a string w denote by $w[i..j]$ the subword of w starting at i and ending at j. Similarly for a long string $\mathcal{W}$ denote $\mathcal{W}[i..j] = \nu(\mathcal{W})[i..j]$.

Denote by $\mathcal{P}$ and $\mathcal{T}$ the descriptions of the pattern P and a text T. P occurs in T at position i iff $T[i..i + |P| - 1] = P$.

The *string matching problem for strings with short description* is:

given $\mathcal{P}$ and $\mathcal{T}$, check if P occurs in T, if "yes" then find any occurrence i.

The size n of the problem is the size $|\mathcal{T}|$ of the description of the text T. Assume $|\mathcal{P}| = m \leq n$.

Our main result is the following theorem.

Theorem 1. *The pattern-matching problem for strings with short descriptions can be solved in polynomial time with respect to the descriptive size.*

A similar problem was considered recently in [2], where strings are described in terms of Lempel-Ziv encoding. The main difference between our results and the results in [2] is that in [2] patterns are assumed to have *explicit* representations, while we consider *implicitly* given patterns with short description.

Example 1. We refer the reader to [6] for definitions of the *Fibonacci* and *Thue-Morse* words. Let $P = F_5$ be the 5th Fibonacci word $abaab$, and $T = T_3$ be the 3rd Thue-Morse word $abbabaab$. We show below short descriptions $\mathcal{F}_5$ and $\mathcal{T}_3$ for these words. An instance of the pattern-matching problem for strings with short description is:

find any occurrence of $\mathcal{F}_5$ in $\mathcal{T}_3$.

An occurrence $i = 4$ of $\mathcal{F}_5$ in $\mathcal{T}_3$ is a solution to this instance.

The 5th Fibonacci word is described by the following program $\mathcal{P}$:

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

The computation of $\mathcal{F}_5$ works as follows.

$$\nu(X_1) = b,\ \ \nu(X_2) = a,\ \ \nu(X_3) = ab,\ \ \nu(X_4) = aba,\ \ \nu(X_5) = abaab.$$

The 3rd Thue-Morse word is described by the following program $\mathcal{T}_3$.

$$\begin{array}{l} X_0 := a; \qquad Y_0 := b; \qquad X_1 := X_0 \cdot Y_0;\ Y_1 := Y_0 \cdot X_0; \\ X_2 := X_1 \cdot Y_1;\ Y_2 := Y_1 \cdot X_1;\ X_3 = X_2 \cdot Y_2 \end{array}$$

The 3rd Thue-Morse word is generated as follows:

$$\begin{array}{l} \nu(X_0) = a, \quad \nu(Y_0) = b, \quad \nu(X_1) = ab, \qquad \nu(Y_1) = ba, \\ \nu(X_2) = abba,\ \nu(Y_2) = baab,\ \nu(X_3) = abbabaab. \end{array}$$

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 T_{200}, despite the fact that *actual* lengths of these strings are astronomic: $|T_{200}| = 2^{200}$ and $|F_{220}| \geq 2^{120}$.

2 Arithmetic Progressions and Euclid's Algorithm

A crucial role in our algorithm is played by 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. Hence $p = |w|$ and $p = 0$ are considered to be periods.

Lemma 2 (periodicity lemma, see [1]).
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 $[1 \ldots N]$ is *succinct* w.r.t. N iff it can be decomposed in at most $\lfloor \log_2(N) \rfloor + 1$ linear sets. For example the set $Periods(aba) = \{0, 2, 3\}$ consists of $\lfloor \log_2(3) \rfloor + 1 = 2$ such sets.

For sets U and W define $U \oplus W = \{i + j \ : \ i \in U, \ j \in W\}$.

Lemma 3 (succinct sets lemma).
The set $Periods(w)$ is succinct w.r.t. $|w|$.

Proof. The proof is by induction with respect to $j = \lfloor \log_2(|w|) \rfloor$. The case $j = 0$ is trivial, one-letter string ($|w| = 1$) has periods 0 and 1 (forming a single progression), hence we have precisely $\lfloor \log_2(|w|) \rfloor + 1$ progressions.

Let $k = \lceil \frac{|w|}{2} \rceil$. It follows directly from Lemma 2 that all periods in $A = Periods(w) \cap [1 \ldots k]$ form a single arithmetic progression, whose step is the greatest common divisor of all of them. Let q be the smallest period larger than k. Then it is easy to see that

$$Periods(w) = A \ \cup \ \{q\} \oplus Periods(w[q + 1..|w|]).$$

Now the claim follows from the inductive assumption, since $\lfloor \log_2(|w| - q) \rfloor < j$ and A is a single progression.

Observe that the structure of $Periods(w)$ corresponds to a *greedy* construction: find the first period p and take the longest progression containing consecutive periods which starts with p, then go to the next period and continue. There are at most $\lfloor \log_2(|w|) \rfloor + 1$ resulting progressions. Assume that we use such type of the representation for sets of periods. Let S_1 be a set of periods of w from $[1..k]$, and S_2 be a set of periods from an interval $[k+1..|w|]$. Then, when adding these sets, it can happen that the last linear set in S_1 continues, with the same step, in S_2, as the first linear set in S_2. We join these two progressions in S and have less linear sets in S.

Generally define the operation $compress(S)$, which for a given set of disjoint linear sets joins any two linear sets (if one is a continuation of the other) wherever it is possible. This operation is important, since we will be often adding succinct sets, and we need also a succinct representation in terms of at most logarithmically many progressions.

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 $Solution(p, U, W)$ 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 4 (application of Euclid 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 $\log(N)$.

Proof. The problem can be easily reduced to the problem:

for given nonnegative integers a, b, c, A, B, find any integer solution (x, y) to the following equation with constraints

$$ax + by = c, \quad (1 \leq x \leq A,\ 1 \leq y \leq B). \tag{1}$$

It is enough to compute a solution in polynomial time with respect to the number of bits of the input constants.

We can assume that a, b are relatively prime, otherwise we can divide the equation by their greatest common divisor.

As a side effect of Euclid algorithm applied to a, b, we obtain integers (not necessarily positive, but with not too many bits) x'_0, y'_0 such that $ax'_0 + by'_0 = 1$. Let $x_0 = cx'_0$, $y_0 = cy'_0$. Then all solutions to the equation (1) are of the form

$$(x, y) = (x_0 + kb, y_0 - ka), \text{ where } k \text{ is an integer parameter.}$$

This defines a line, and we have to find any integer point in the rectangle $\{(i, j) : 1 \leq i \leq A,\ 1 \leq j \leq B\}$ which is *hitten* by this line. This can be done in polynomial time using operations *div* and *mod* on integers with polynomial number of bits. We refer for details to [5] (see page 325 and Exercise 14 on page 327).

3 The Pattern-Matching Algorithm

Let us fix the pattern $P = \nu(\mathcal{P})$, the length of P is M and the length of the text $T = \nu(\mathcal{T})$ is N. Observe that $N = O(2^n)$, hence all positions in T can be written using $O(n)$ bits.

Let X be a string (long or short) of the length K. Then define:

$$\mathit{Prefs}(X) = \{1 \leq i \leq K \ : \ X[K - i + 1..K] \text{ is a prefix of } P\}.$$
$$\mathit{Suffs}(X) = \{1 \leq i \leq K \ : \ X[1..i] \text{ is a suffix of } P\}.$$

Observation 1
Let $\mathcal{P}$, $\mathcal{A}$, $\mathcal{B}$ be long strings, then $\mathcal{P}$ occurs in $\mathcal{A} \cdot \mathcal{B}$ iff:
(1) $\mathcal{P}$ occurs in $\mathcal{A}$ or $\mathcal{P}$ occurs in $\mathcal{B}$;
(2) or $|P| \in \mathit{Prefs}(\mathcal{A}) \oplus \mathit{Suffs}(\mathcal{B})$.

Define the operations of the *prefix-extension* and *suffix-extension*. For a long or a short word X define

$$\mathit{PrefExt}(S, X) = \{i + |X| \ : \ i \in S \text{ and } P[1..i] \cdot X \text{ is a prefix of } P\}.$$
$$\mathit{SuffExt}(S, X) = \{i + |X| \ : \ i \in S \text{ and } X \cdot P[M - i + 1..M] \text{ is a suffix of } P\}.$$

Assume the straight line program $\mathcal{T}$ defining the text T is using variables X_1, X_2, ..., X_n. Each of variables X_i corresponds to a program $\mathcal{X}_i$ which computes X_i. We denote

$$\mathit{SUFF}[i] = \mathit{Suffs}(\mathcal{X}_i), \quad \mathit{PREF}[i] = \mathit{Prefs}(\mathcal{X}_i).$$

Observe that these tables depend on the pattern P, however it is convenient to assume further that P is fixed. We are now able to give a sketch of the whole structure of the algorithm. Assume that the lengths of all strings described by $\mathcal{X}_k$'s are computed (it can be easily done in polynomial time).

```
ALGORITHM PATTERN_MATCHING ;
for k = 1 to n do
    if |ν(X_k)| ≤ n then {X_k can be treated as an explicit string}
        test by classical methods an occurrence of P in X_k;
        if there is an occurrence of P in X_k then report it and STOP
            else compute PREF[k], SUFF[k] by classical methods;
    else {assume X_k = X_i · X_j for i, j < k }
        pos := Solution(|P|, PREF[i], SUFF[j]);
        if pos ≠ 0 then report an occurrence and STOP
            else begin
                U := PrefExt(PREF[i], X_j) ∪ PREF[j];
                V := SuffExt(SUFF[j], X_i) ∪ SUFF[i];
                PREF[k] := compress(U); SUFF[k] := compress(V);
            end
```

Let k be the first position in $PREF[i]$, then all the other positions in $PREF[i]$ are of the form $k + p'$, where p' is a period of $P[1..k]$. Hence Lemma 3 implies directly the following fact.

Lemma 5. *The sets $SUFF[i]$ and $PREF[j]$ are succinct, for any $1 \leq i, j \leq n$.*

For a sequence of long strings $\gamma = \mathcal{X}_1, \ldots, \mathcal{X}_p$ define $\nu(\gamma) = \nu(\mathcal{X}_1)\nu(\mathcal{X}_2)\ldots\nu(\mathcal{X}_p)$. We omit the proof of the following fact. The proof employs the algorithm from [7] as a subroutine, and a kind of binary search in $[1 \ldots N]$.

Lemma 6 (subword-equality).
(a) *For two sequences of long strings $\gamma_1 = \mathcal{X}_1, \ldots, \mathcal{X}_p$ and $\gamma_2 = \mathcal{Y}_1, \ldots, \mathcal{Y}_q$ we can test equality $\nu(\gamma_1) = \nu(\gamma_2)$ in polynomial time with respect to the total size of all $\mathcal{X}_i$'s and $\mathcal{Y}_j$'s.*
(b) *For two long strings $\mathcal{X}, \mathcal{Y}$ and integers i, j, k, l we can test the equality $\mathcal{X}[i..j] = \mathcal{Y}[k..l]$, and find the first mismatch (if there is any) in polynomial time with respect to the size of description.*

Let us call the algorithms implied by the lemma the *equality-test algorithms*.

Our key lemma says that the operations *PrefExt* and *SuffExt* are *feasible*. 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.

[htbp]

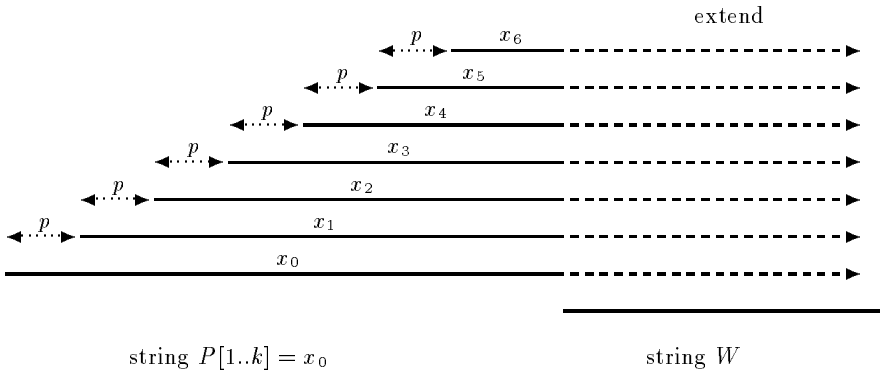

Fig. 1. The operation $PrefExt(S, W)$, where $S = \{|x_0|, |x_1|, \ldots, |x_6|\}$.

Lemma 7 (key lemma).
Assume W is a long word, 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 $x_i = P[1..t_i]$, $0 \leq i \leq s$, are suffixes of $P[1..k]$. Then the representation of $PrefExt(S, W)$ can be computed in polynomial time.

Proof. Assume the sequence $t_0, t_1, \ldots, t_s$ is decreasing. We need to compute all possible continuation of x_i's in P which match W, see Figure 1. Denote $y_i = P[1..|x_i| + |W|]$ and $Z = P[1..k] \cdot W$. Hence our aim is to find all i's such that y_i is a suffix of Z, $(0 \leq i \leq s)$. 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 $x[mismatch] \neq x[mismatch - p]$. We can compute the first mismatch using an equality-test algorithm from Lemma 6. There are four basic cases:

Case A: there is no mismatch in Z but there is a mismatch in y_0.
Then good indices are all $i \geq r$, where r is the first index such that y_r contains no mismatch at all. (We have $r = 4$ in Figure 2 (case A)).

Case B: there is a mismatch in Z and y_0.
Then the only possible good index i is such that the first mismatch in y_i is exactly over the first mismatch in Z. See Figure 2 (case B), where the only good index is $i = 2$. We can easily calculate such i, it is also possible that there is no good i in this case.

Case C: there is no mismatch in Z or y_0.
Then all indices i are good.

Case D: there is a mismatch in Z but not in y_0.
Then none of indices i is 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

[htb]

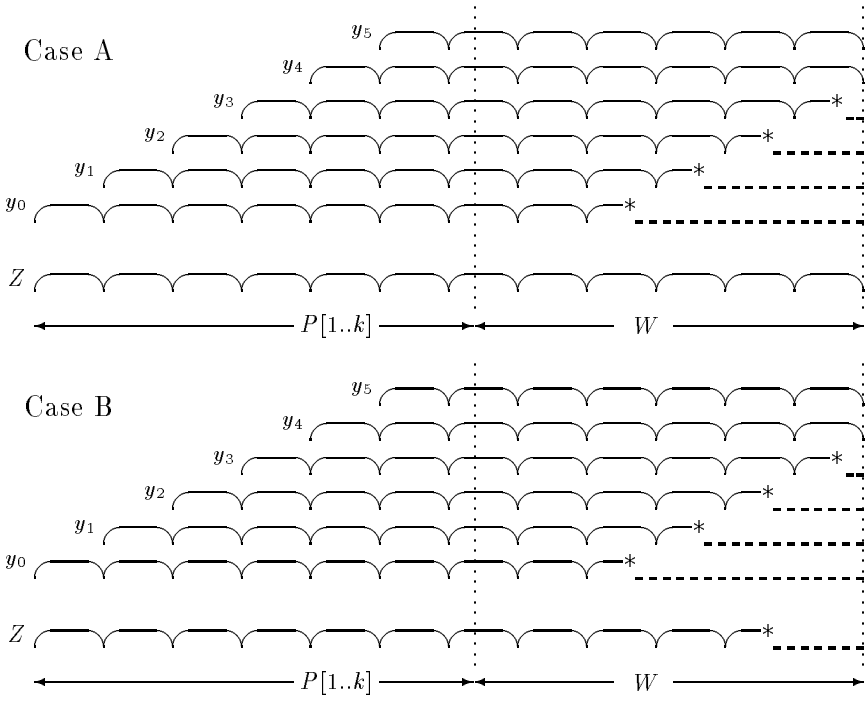

Fig. 2. Two cases: $Z = P[1..k] \cdot W$ has no mismatch in Case A, but Z has mismatch in Case B. $y_i = P[1..|x_i| + |W|]$. "$*$" denotes the mismatch to the period.

required output) of integers $\{|y_i| \ : i \text{ is a good index } \}$ is linear. This completes the proof.

If the sets $SUFF[j], PREF[j]$ 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 $PREF[i]$ and $SUFF[i]$ in polynomial time using polynomially many time the algorithm from Lemma 7 to each of these linear sets. In this way we have shown that the algorithm *PATTERN_MATCHING* works in polynomial time. This completes the proof of our main result (Theorem 1).

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

Theorem 8. *Assume $\mathcal{X}$ is a string given by its description of size n. Then we can compute in polynomial time a polynomial size representation of set Periods$(\nu(\mathcal{X}))$. The representation consists of a linear number of linear sets.*

Proof. Use the algorithm *PATTERN_MATCHING* with the long pattern $\mathcal{P} = \mathcal{X}$ and the long text $\mathcal{T} = \mathcal{X}$. As a side effect we compute all suffixes of $\mathcal{T}$ which are prefixes of $\mathcal{P}$. This determines easily all periods.

4 An NP-Complete Version of Pattern-Matching

We start with a problem which has a particularly simple polynomial time algorithm, next we show that an extension of this problem is NP-complete.

The pattern–matching algorithm is much simpler if the pattern P is an *explicit word* of length $m = O(n)$, where n is the descriptive size of the *long* text $\mathcal{T}$. Let X_i be the variables of a program describing $\mathcal{T}$.

Denote $ShortVar = \{X_i : |\nu(X_i)| \leq m\}$. For each variable $X_i \in ShortVar$ we can compute its value by simply simulating the given straight-line program. We need $O(n \cdot m)$ time for all X_i's together.

The algorithm which looks for the explicit pattern P by searching *only* inside all words X_i (incorrectly assuming the pattern is contained totally in some of X_i's) is incorrect.

For a language L over the alphabet $ShortVar$ define $\nu(L) = \{\nu(\gamma) \ : \gamma \in L\}$.

Lemma 9. *Assume an (explicit) word P is of the real size $m = O(n)$. Then there is a nondeterministic finite automaton A accepting a language L over $ShortVar$ such that:*

the pattern P occurs in $\mathcal{T}$ iff $P \in \nu(L)$.

The constructed nondeterministic automaton A has $O(n)$ states.

Proof. We omit the proof.

We can replace each edge labeled X_i of the automaton A from the lemma above by $|\nu(X_i)|$ edges "spelling" the word $\nu(X_i)$. Then the automaton grows by a factor $O(m)$. The new automaton A' has the size $O(n^2)$. It can be applied to test if P occurs in $\mathcal{T}$ by simulating A' on P. A standard method can be used to test if a nondeterministic automaton accepts a text. This proves the following theorem.

Theorem 10. *Assume we have an (explicit) pattern P given explicitly of size $m = O(n)$ and a string $\mathcal{T}$ given by its description of size n. Then we can test if P occurs in $\mathcal{T}$ in $O(n^3)$ time.*

We show that the automata theoretic approach which was used above (and which corresponds to regular expressions) does not work if the pattern is a long string.

Let Var be a set of variables in some straight-line program of length n, and $\mathcal{P}$ be a long pattern (given by a straight-line program of length $m \leq n$). We consider the *regular-expression-matching* problem for shortly described strings defined as follows:

given a regular expression W over *Var* of size $O(n)$
test if $\nu(\mathcal{P}) \in \nu(W)$,

where $\nu(W) = \nu(L)$, and L is the language described by expression W.

Theorem 11. *The regular-expression-matching problem for shortly described strings is NP-complete, even if expressions do not contain operation *, nor empty strings and the alphabet Σ (for strings which are values of variables) is unary.*

Proof. The proof is a reduction from the SUBSET SUM problem defined as follows:

Input instance: Finite set $A = \{a_1, a_2, \ldots, a_n\}$ of integers and an integer K. The size of the input is the number of bits needed for the description.
Question: Is there a subset $A' \subseteq A$ such that the sum of the elements in A' is exactly K?

The problem SUBSET SUM is NP-complete, see [4], and [3], pp. 223. We can construct easily a straight-line program such that $\nu(X_i) = 1^{a_i}$ and $P = 1^K$. Then the SUBSET SUM problem is reduced to the membership:

$$\nu(\mathcal{P}) \in \nu((X_1 \cup \varepsilon) \cdot (X_2 \cup \varepsilon) \cdots (X_n \cup \varepsilon)).$$

The empty string ε can be easily eliminated by rescaling numbers and replacing ε by a single letter 1. This completes the proof.

References

1. M. Crochemore and W. Rytter, Text Algorithms, Oxford University Press, New York (1994).
2. M. Farach and M. Thorup, "String-matching in Lempel-Ziv compressed strings", to appear in Proc. 27th ACM STOC (1995).
3. M.R. Garey and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman (1979).
4. R.M. Karp, "Reducibility among combinatorial problems", in R.E. Miller and J.W. Thatcher (eds.), Complexity of Computer Computations, Plemum Press, New York, pp.85–103 (1972).
5. D. Knuth, *The Art of Computing, Vol. II: Seminumerical Algorithms. Second edition.* Addison-Wesley (1981).
6. M. Lothaire, *Combinatorics on Words.* Addison-Wesley (1993).
7. W. Plandowski, "Testing equivalence of morphisms on context-free languages", ESA'94, Lecture Notes in Computer Science 855, Springer-Verlag, 460–470 (1994).