Nordic Journal of Computing 4(1997), 172–186.

AN EFFICIENT PATTERN-MATCHING ALGORITHM FOR STRINGS WITH SHORT DESCRIPTIONS

MAREK KARPINSKI
Department of Computer Science,
University of Bonn,
53117 Bonn, Germany
marek@cs.uni-bonn.de

WOJCIECH RYTTER
Institute of Informatics,
Warsaw University,
ul. Banacha 2,
02-097 Warszawa, Poland
rytter@mimuw.edu.pl

AYUMI SHINOHARA
Department of Informatics,
Kyushu University 33,
Fukuoka 812-81, Japan
ayumi@i.kyushu-u.ac.jp

Abstract. We investigate the time complexity of the pattern matching problem for strings which are succinctly described in terms of straight-line programs, in which the constants are symbols and the only operation is the concatenation. Most strings of descriptive size n are of exponential length with respect to n. A complicated algorithm for the *equality-test* (testing if two succinctly described strings are the same) in $O(n^4)$ time was constructed by Plandowski. Our main aim is to extend this result and we show that a much stronger problem of the *pattern-matching* for succinctly described strings can be solved with similar complexity: in $O(n^4 \log n)$ time. However, Plandowski's algorithm is not used in this paper, and our algorithm gives independent constructive proof. The crucial point in our algorithm is the succinct representation of all periods of a (possibly long) string described in this manner. We also show a (rather straightforward) result that a very simple extension of the pattern-matching problem for shortly described strings is NP-complete.

CR Classification: E.4, F.2.2

Key words: pattern matching, text search, compression, straight-line program

1. Introduction

The *pattern-matching* problem is the central algorithmic problem related to *texts* [see Crochemore and Rytter 1994]. Recently the problem was considered in a *compressed setting* [see Farach and Thorup 1995]. We use a simpler type of compression in terms of straight-line programs (grammars, or recurrences), in which the constants are symbols and the only operation is the concatenation. Examples of typical words described in this way are *Fibonacci words* and *Thue-Morse* words. Such a description (compression) is *sufficiently powerful* that several interesting strings can be described *succinctly*; at the same time, it turns out to be *sufficiently simple* that basic

Received July 1995. Revised October 1996.

queries such as equality testing and (as we show here) pattern matching can be answered efficiently.

Our *main aim* is to extend the result of Plandowski [1994] from the equality-testing problem to the pattern-matching problem. However, we do not use Plandowski's algorithm in the paper, and our algorithm gives independent constructive proof of the result obtained by Plandowski.

The main difference between our results and the results of Farach and Thorup [1995] is that in the latter paper patterns are assumed to have *explicit* representations, while we allow patterns to be given *implicitly* by a short description, so our results are rather incomparable.

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.

Denote by R the string which is the value of the last variable X_n after the execution of the program $\mathcal{R}$. 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. We identify variables with their values in the sequel. R is called a *string with short description*, since most $|R|$ is very long (exponentially) with respect to its descriptive size $|\mathcal{R}|$. For a string w denote by $w[i..j]$ the subword of w starting at i and ending at j.

Denote by $\mathcal{P}$ and $\mathcal{T}$ the descriptions of a pattern P and a text T. We say that P *occurs in* T *at position* i if $T[i..i + |P| - 1] = P$. The *pattern matching problem for strings with short descriptions* 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* $O(n^4 \log n)$ *time.*

EXAMPLE 1. Let us consider the following straight-line programs $\mathcal{T}$ and $\mathcal{P}$. The program $\mathcal{T}$ describes the 8*th Fibonacci word*, [see Lothaire 1983].

$$\begin{array}{rlcrl}
\mathcal{T}: & X_1 = \mathsf{b}; & \qquad & \mathcal{P}: & Y_1 = \mathsf{b}; \\
& X_2 = \mathsf{a}; & & & Y_2 = \mathsf{a}; \\
& X_3 = X_2 \cdot X_1; & & & Y_3 = Y_2 \cdot Y_1; \\
& X_4 = X_3 \cdot X_2; & & & Y_4 = Y_2 \cdot Y_3; \\
& X_5 = X_4 \cdot X_3; & & & Y_5 = Y_3 \cdot Y_2; \\
& X_6 = X_5 \cdot X_4; & & & Y_6 = Y_4 \cdot Y_4; \\
& X_7 = X_6 \cdot X_5; & & & Y_7 = Y_6 \cdot Y_5. \\
& X_8 = X_7 \cdot X_6, & & &
\end{array}$$

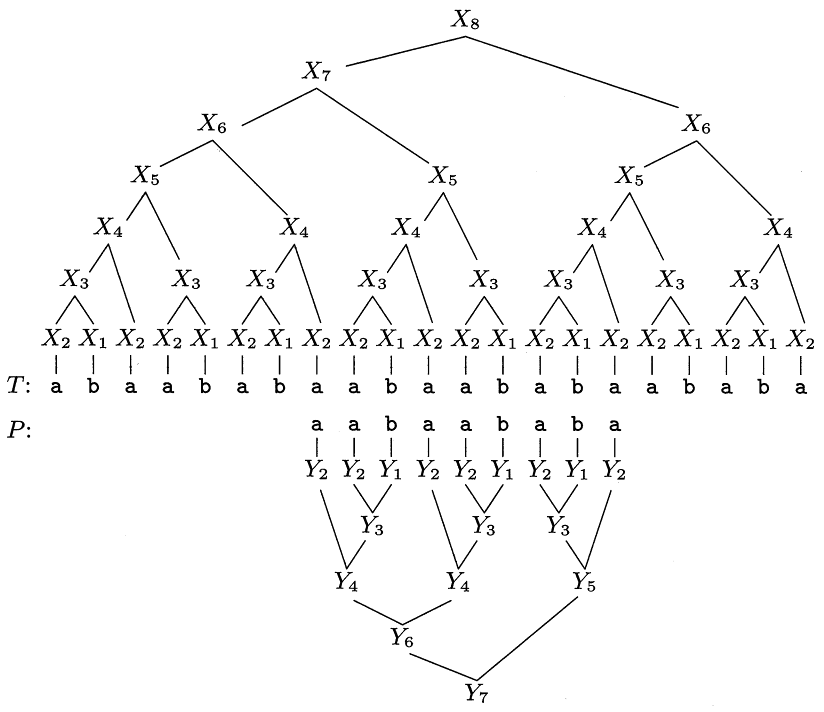

Fig. 1: The evaluation trees for the text T (top) and the pattern P (reverse at bottom), there is an occurrence of the pattern P starting at position 8 of T.

We can see that

$$\begin{aligned} T &= X_8 = \mathsf{abaababaabaababaababa}, \\ P &= Y_7 = \mathsf{aabaababa}, \end{aligned}$$

as shown in Fig. 1. An occurrence $i = 8$ of P in T is a solution to this instance. As we show in the subsequent sections, we can find such an occurrence *without expanding the strings explicitly.* Remark that the problem we deal with is not simply the classical *tree matching problem*, although the text and the pattern have a natural representation as trees. In fact, the evaluation tree for the pattern P does not match any subtree of the evaluation tree for the text T in Fig. 1.

A preliminary version of this paper was presented at Karpinski *et al.* [1995].

2. The basic idea of the algorithm

For two strings x and y, we define the set of overlaps between x and y by

$$Overlaps(x, y) = \{k > 0 \ : \ x[|x| - k + 1..|x|] = y[1..k]\}.$$

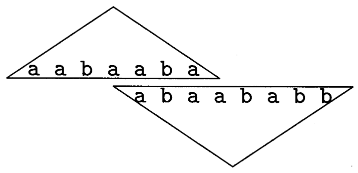

Fig. 2: *Overlaps*("aabaaba", "abaababb") $\ni 3$

For example, *Overlaps*("aabaaba", "abaababb") $= \{1, 3, 6\}$, see Fig. 2. For two sets U and V of integers, we define $U \oplus V = \{i + j \mid i \in U, j \in V\}$.

Without loss of generality, we assume that the given straight-line programs $\mathcal{P}$ and $\mathcal{T}$ contain no *redundant* assignment $X_i = expr_i$ $(1 \le i < n)$ that is never referred, since such a redundant assignment can be deleted in linear time.

Both algorithms in this paper and in Karpinski *et al.* [1995] are based on the following observation (see Fig. 3):

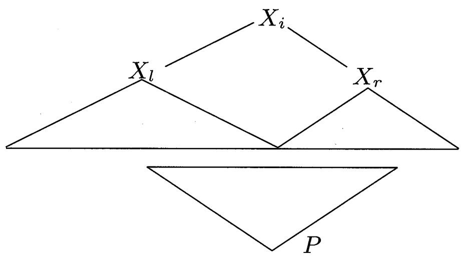

Fig. 3: P appears in X_i

OBSERVATION 1.
P occurs in T if and only if $\mathcal{T}$ contains an assignment statement $X_i = X_l \cdot X_r$ such that for some t, $P[1..t]$ is a suffix of X_l and $P[t+1..|P|]$ is a prefix of X_r; that is, $|P| \in Overlaps(X_l, P) \oplus Overlaps(P, X_r)$.

The algorithm scheme consists of the following two phases:

Preprocess: For each variable X_i in $\mathcal{T}$,
compute $Overlaps(X_i, P)$ and $Overlaps(P, X_i)$.

Pattern detection: Find X_i such that $X_i = X_l \cdot X_r$ and
$|P| \in Overlaps(X_l, P) \oplus Overlaps(P, X_r)$.

The difficulty we must overcome is that the size of the set $Overlaps(X_i, P)$ (and $Overlaps(P, X_i)$ also) grows exponentially with respect to the size

$n = |\mathcal{T}|$ of an instance. We need to develop some good representation of the set in order to construct a polynomial-time algorithm for the pattern matching problem. In this paper, we represent the set *Overlaps*(X_i, P) by $O(n)$ arithmetic progressions by clever use of *periodicities* in strings. Using this data structure, the pattern detection phase can be done in $O(n^4)$ time, as we will show in Section 4. According to the preprocess phase, Karpinski *et al.* [1995] shows an $O(n^7)$ time algorithm which repeatedly calls the equality testing procedure due to Plandowski [1994]. In this paper, we realize $O(n^4 \log n)$ time algorithm for the preprocess, which dominates the total time complexity, by computing *Overlaps*(X, Y) for each pair of variables in $\mathcal{T}$ and $\mathcal{P}$ in *bottom-up* manner.

3. Preprocess

In the preprocess phase, we compute *Overlaps*(X, Y) for all pairs X and Y of variables appeared in given straight-line programs $\mathcal{T}$ or $\mathcal{P}$, and memorize all of them using only $O(n^3)$ space. In this section, we introduce the overlap table and associated procedures.

3.1 Periodicity and succinct representation

First we state some facts about periodicities useful in the construction of the succinct representation.

A nonnegative integer p is a *period* of a nonempty string w if $w[i] = w[i-p]$, whenever both sides are defined. *Periods*(w) denotes the set of all periods of w. For example, *Periods*("aabaabaabaa") $= \{0, 3, 6, 9, 10, 11\}$.

LEMMA 1. (SEE CROCHEMORE AND RYTTER 1994, PP. 29) *If w has two periods p and q such that $p + q \le |w|$ then* $\gcd(p, q)$ *is a period of w, where* gcd *means the "greatest common divisor".*

We say that a set of nonnegative integers from $\{0, 1, \ldots, N\}$ is *succinct* with respect to N if it can be decomposed into at most $\lfloor \log_2 N \rfloor + 1$ arithmetic progressions. For example, the set *Periods*("aba") $= \{0, 2, 3\}$ consists of $\lfloor \log_2 3 \rfloor + 1 = 2$ arithmetic progressions. The following fact is a consequence of Lemma 1.

LEMMA 2. *The set Periods(w) is succinct with respect to $|w|$.*

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 1 that all periods in $U =$ *Periods*$(w) \cap \{0, 1, \ldots k\}$ 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

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

	X_1	X_2	X_3	X_4	X_5	X_6	X_7	X_8
X_1	$\{1\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
X_2	$\emptyset$	$\{1\}$	$\{1\}$	$\{1\}$	$\{1\}$	$\{1\}$	$\{1\}$	$\{1\}$
X_3	$\{1\}$	$\emptyset$	$\{2\}$	$\{2\}$	$\{2\}$	$\{2\}$	$\{2\}$	$\{2\}$
X_4	$\emptyset$	$\{1\}$	$\{1\}$	$\{1,3\}$	$\{1,3\}$	$\{1,3\}$	$\{1,3\}$	$\{1,3\}$
X_5	$\{1\}$	$\emptyset$	$\{2\}$	$\{2\}$	$\{2,5\}$	$\{2,5\}$	$\{2,5\}$	$\{2,5\}$
X_6	$\emptyset$	$\{1\}$	$\{1\}$	$\{1,3\}$	$\{1,3\}$	$\{1,3,8\}$	$\{1,3,8\}$	$\{1,3,8\}$
X_7	$\{1\}$	$\emptyset$	$\{2\}$	$\{2\}$	$\{2,5\}$	$\{2,5\}$	$\{2,5,13\}$	$\{2,5,13\}$
X_8	$\emptyset$	$\{1\}$	$\{1\}$	$\{1,3\}$	$\{1,3\}$	$\{1,3,8\}$	$\{1,3,8\}$	$\{1,3,8,21\}$

TABLE I: The overlap table of the straight-line program $\mathcal{T}$ in Example 1.

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

3.2 Compressed overlap table

For a straight-line program $\mathcal{R}$ of length n, we define an *overlap table* for $\mathcal{R}$ as the $n \times n$ array, where the content at column X_i and row X_j is the set *Overlaps*(X_i, X_j). Remark that the cardinality of the set *Overlaps*(X_i, X_j) might be $\Omega(2^n)$.

EXAMPLE 2. Let us consider the straight-line program $\mathcal{T}$ for the 8*th Fibonacci word* in Example 1. Table I shows its overlap table.

The *overlap query* is the question of the form: "$k \in$ *Overlaps*(X, Y) ?", for an integer k and variables X and Y. We now develop a data structure *compressed overlap table* of a straight-line program $\mathcal{R}$, which requires only $O(n^3)$ space to represent all (potentially exponentially many) overlaps between variables of $\mathcal{R}$, and allows to answer each *overlap query* in $O(\log n)$ time.

A *compressed overlap table OV* is an overlap table where each content of the set *Overlaps*(X, Y) is succinctly represented using $O(n)$ arithmetic progressions as follows.

First of all, remark that the set *Overlaps*(X, Y) can be expressed by the pair $o_{\max} = \max\{i : i \in \textit{Overlaps}(X, Y)\}$ and *Periods*$(Y[1..o_{\max}])$, since *Overlaps*$(X, Y) = \{o_{\max} - i : i \in \textit{Periods}(Y[1..o_{\max}])\} - \{0\}$. According to Lemma 2, the set *Periods*$(Y[1..o_{\max}])$ consists of at most n arithmetic progressions. Actually, we store the set of periods as follows (see Fig. 4): Let a_1 be the smallest period, that is always zero. Let d_1 be the difference between a_1 and the next period in increasing order. For $i > 1$, let a_i be the first period which is not in the set $\cup_{k=0}^{i-1}\{a_k + j \cdot d_k : j \geq 0\}$, and d_i is the difference between a_i and the previous period. Each pair $\langle a_i, d_i \rangle$ represents the segment $\{a_i + jd_i : 0 \leq j \text{ and } a_i + jd_i < a_{i+1}\}$ of the periods. We keep

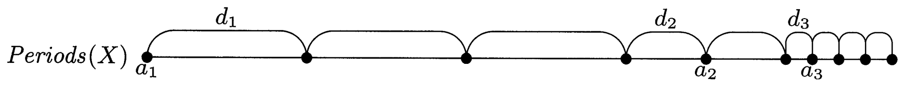

Fig. 4: The set $Periods(X)$ is stored as the pairs $\langle a_1, d_1\rangle$, $\langle a_2, d_2\rangle$, ..., where dots • denote the periods of X.

these segments $\{\langle a_i, d_i\rangle\}$ in a sorted order with respect to a_i's so that any overlap query can be answered in short time.

LEMMA 3. *If the set of all overlaps between two variables X and Y is computed and represented using compressed overlap table then each overlap query between X and Y can be answered in $O(\log n)$ time.*

PROOF. First, we find the segment to which given value may belong, by using binary search with respect to a_i's. It takes $O(\log n)$ time, since there are at most n segments. Then we can determine whether the value is in the progression or not in constant time by simple arithmetic operations. □

3.3 Operation Compress

Assume we have a (possibly redundant) representation of the set V of all overlaps between two variables X_i and X_j in terms of of progressions corresponding to periods of $Periods(X_j[1..k])$ for some k. Such a situation arises when we tries to construct V by merging two sets, which will be explained in Section 3.5. Then for each pair of progressions we check if one is contained in the other and whenever this happens we remove a *redundant* progression. It takes $O(n^2)$ time. The resulting set is denoted by $Compress(V)$.

3.4 First mismatch and period continuation

Assume $\mathcal{R}$ is a given straight-line program, whose variables are $X_1, \ldots, X_n$. We say that X_i *precedes* X_j if $i < j$. Assume $|Y| \geq k$. Define $FirstMismatch(X, Y, k)$ as the first mismatch (from left) which is a witness to the fact that $k \notin Overlaps(X, Y)$. More formally,

$$FirstMismatch(X, Y, k) = \min\{i > 0 \;:\; X[|X| - k + i] \neq Y[i]\},$$

for $0 \leq k \leq |X|$. If there is no such i, the value of $FirstMismatch(X, Y, k)$ is 0.

THEOREM 2. *Assume A and B are two variables of a given straight-line program of length n. Then the value of $FirstMismatch(A, B, k)$ can be computed with $O(n)$ overlap queries between variables which precede A or B.*

PROOF. Consider the evaluation trees for variables A and B. The internal nodes can be identified with corresponding variables. We use a kind of *binary*

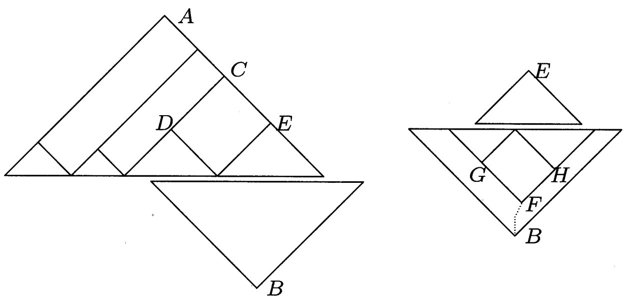

Fig. 5: Looking for the first mismatch in *Overlaps*(A, B).

```
function FirstMis(variable $X_i, X_j$; int $k$) : integer    /* $0 < k \le |X_i|$ */
    if ( $X_i$ is a terminal ) then
        if ( $X_j$ is a terminal ) then
            if ( $X_i = X_j$ ) then return 0 else return 1
        else /* Assume $X_j = X_L \cdot X_R$ */
            if ( $k \in Overlaps(X_i, X_L)$ ) then return 0 else return 1
    else begin /* Assume $X_i = X_l \cdot X_r$ */
        if ( $|X_i| - k \ge |X_l|$ ) then return $FirstMis(X_r, X_j, k)$
        else if $(k - |X_j| \ge |X_r|)$ then return $FirstMis(X_l, X_j, k - |X_r|)$
        else if ( $k - |X_r| \notin Overlaps(X_l, X_j)$ ) then return $FirstMis(X_l, X_j, k - |X_r|)$
        else begin
            $mis := FirstMis(X_j, X_r, |X_j| + |X_r| - k)$;
            if ( $mis = 0$ ) then return 0 else return $(k - |X_r| + mis)$
        end
    end
```

Fig. 6: A pseudo-code which computes the FirstMismatch.

search going down alternatively in the first or the second tree. Assume we are to compute the first mismatch in the overlap between A and B, see Fig. 5. By calculating the length of each variable, we go down the tree with root A to find the first node C such that its left subtree makes an overlap with B, see Fig. 5 (on the left). Then we make an overlap query between B and D. If the answer is "no", then the first mismatch should be in this part, and we search recursively for a mismatch in the overlap between D and B. Otherwise, we go down (up on the figure) from the root in the tree rooted at B to find the first node F whose sons G and H overlap the tree rooted at E. Then we make an overlap query between G and E. If the answer is "no" then we search recursively for a mismatch in this overlap. Otherwise we search recursively in the overlap between H and E.

In this way after a constant number of overlap queries we go down (towards the leaves) in one of the trees. The height of the trees is $O(n)$, hence the number of queries is $O(n)$. This completes the proof. A pseudo-code which computes the FirstMismatch is given in Fig. 6. □

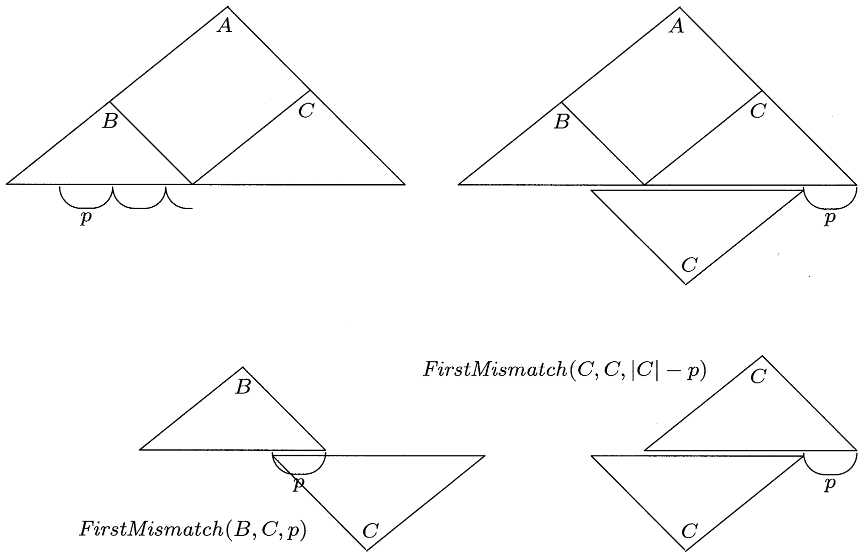

Fig. 7: We know that in A there is the period p in the part B. The first mismatch to the continuation of this periodicity in C is found by two applications of searching first mismatches in overlaps.

THEOREM 3. *Assume the compressed overlap table of all variables preceding A is constructed, and there is a period p in the B part of A, where $A = B \cdot C$. Then the first mismatch to the continuation of the period p in C can be computed in $O(n \log n)$ time.*

PROOF. We refer to Fig. 7. First we search for the mismatch in the overlap between B and C, using the algorithm *FirstMismatch*(B, C, p) in Theorem 2. Then, if there was no mismatch, we search in the overlap between C and C by *FirstMismatch*$(C, C, |C| - p)$. By Lemma 3 and Theorem 2, the running time is $O(n \log n)$. □

3.5 Computing the compressed overlap table of $\mathcal{P}$ and $\mathcal{T}$

Let $\mathcal{R}$ be the *concatenation* of straight-line programs $\mathcal{P}$ and $\mathcal{T}$ together. We show how to compute efficiently the compressed overlap table for all variables in $\mathcal{R}$. Assume the variables are $X_1, \ldots, X_n$. The computation is *bottom-up.* For each pair of terminal variables X and Y, the algorithm first computes $OV[X, Y]$ in a *naive* way, Then it computes the elements of the table OV according to the order presented in Fig. 8. The basic auxiliary operation is the *prefix extension.* Assume that $U \subseteq \{1, \ldots, N\}$. Define

$$PrefExt(U, A, B) = \{k + |B| \ : \ k \in U, \ \ A[1..k] \cdot B \text{ is a prefix of } A\}.$$

The algorithm is based on the following obvious fact:

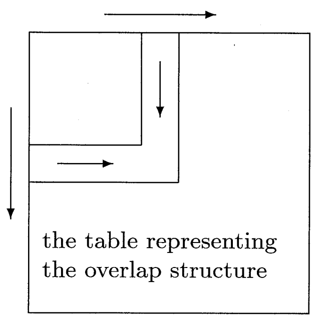

Fig. 8: The order of processing elements of the table OV.

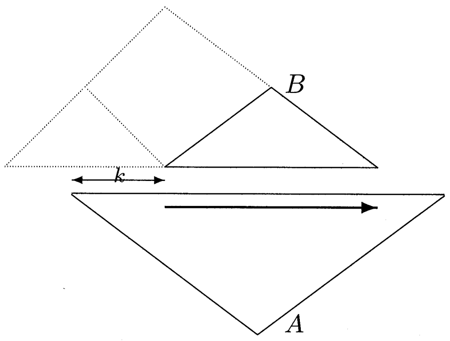

Fig. 9: $k + |B| \in \mathit{PrefExt}(U, A, B)$ if and only if the common parts of A and B agree.

OBSERVATION 2. Assume that $X_i = X_l \cdot X_r$ and $U := OV[X_l, X_j]$ and $W := OV[X_r, X_j]$. Then

$$OV[X_i, X_j] := \mathit{Compress}(\mathit{PrefExt}(U, X_j, X_r) \cup W).$$

```
ALGORITHM Compute_Overlap_Table ;
    compute $OV[X_i, X_j]$ for each pair of terminal variables;
    for $(i, j)$ in the order shown in Fig. 8 do
        begin
            /* Assume $X_i = X_l \cdot X_r$ */
            /* computation of $OV[X_i, X_j]$ */
            $U := OV[X_l, X_j]$;
            $U := \mathit{PrefExt}(U, X_j, X_r)$;
            $W := OV[X_r, X_j]$;
            $OV[X_i, X_j] := \mathit{Compress}(U \cup W)$;
        end
```

The next lemma says that the operations *PrefExt* is efficient for a single progression.

LEMMA 4. *Assume A and B are variables and the compressed overlap table for variables which precede A or B is computed. Let $S = \{t_0, t_1, \ldots, t_s\} \subseteq \{1, \ldots, k\}$ be an arithmetic progression given by its succinct representation, where $t_0 = k \leq |A|$ and strings $x_i = A[1..t_i]$, $0 \leq i \leq s$, are suffixes of $A[1..k]$. Then the representation of PrefExt(S, A, B) can be computed in $O(n \log n)$ 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 A which match B, see Fig. 9. Denote $y_i = A[1..|x_i| + |B|]$ and $Z = A[1..k] \cdot B$. 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. Let $p = t_1 - t_0$ be the step of the linear set S. Then p is the period of $A[1..k]$.

We can compute the first mismatch to the continuation of periodicity p in Z and in y_0 using the algorithm from Lemma 3. There are four basic cases:

Case A: there is no mismatch in Z but there is a mismatch for the periodicity p 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 Fig. 10 (case A).

Case B: there is a mismatch in Z and a mismatch in y_0.
The good index, if any, could only be i such that the first mismatch in y_i is exactly over the first mismatch in Z. See Fig. 10 (case B), where the only good index is $i = 2$. We can easily calculate such i.

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 required output) of integers $\{|y_i| \; : i \text{ is a good index } \}$ forms an arithmetic progression. This completes the proof. □

The set U in the algorithm *Compute_Overlap_Table* consists of $O(n)$ arithmetic progressions. Hence for each pair (X_i, X_j) we perform $O(n)$ operations *PrefExt* applied to an arithmetic progressions. Altogether we do $O(n^3)$ such operations, hence the total time is $O(n^4 \log n)$. This implies the main result of this section:

THEOREM 4. *The compressed overlap table for a given straight-line program $\mathcal{R}$ can be computed in $O(n^4 \log n)$ time.*

As a side effect of Theorem 4 we can compute the set of all periods for strings with short descriptions.

COROLLARY 1. *Assume T is a string given by its description $\mathcal{T}$ of size n. Then we can compute in $O(n^4 \log n)$ time a linear size representation of the set Periods(T).*

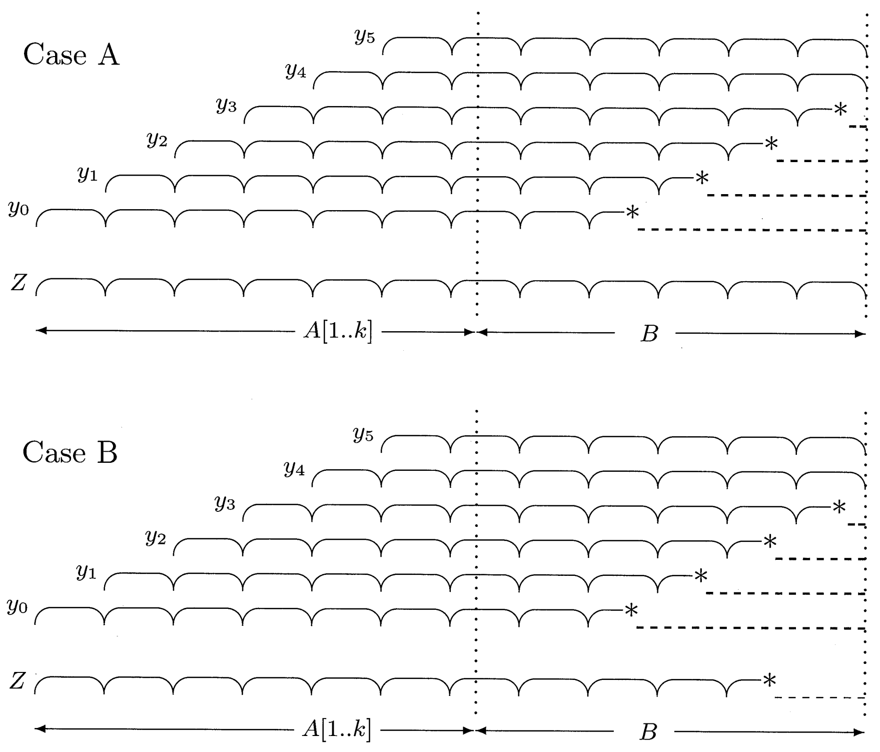

Fig. 10: Two cases: $Z = A[1..k] \cdot B$ has (has no) mismatch, $y_i = A[1..|x_i| + |B|]$.

4. The Pattern-Matching Algorithm

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. For two sets U and V of integers and an integer p, we 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 5.
Assume that two arithmetic progressions $U, W \subseteq \{1, \ldots, N\}$ are given by their descriptions. Then for a given number $c \in \{1, \ldots, N\}$ we can compute $Solution(c, U, W)$ in $O(n^2)$ time, where $n = \log N$.

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

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

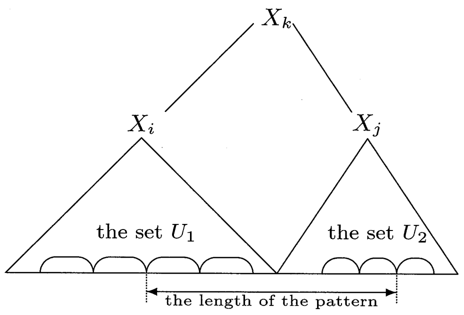

Fig. 11: $U_1 = OV[X_i, P]$ and $U_2 = OV[P, X_j]$, the algorithm find positions in these sets whose difference is the length of the pattern.

It is enough to compute a solution in $O(n^2)$ 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 $\{(i, j) : 1 \le i \le A,\ 1 \le j \le B\}$ which is *hit* by this line. This can be done in $O(n^2)$ time using operations *div* and *mod* on integers. We refer for details to Knuth [1981] (see page 325 and Exercise 14 on page 327). □

```
**ALGORITHM** *Pattern_Matching* ;
*Compute_Overlap_Table* ;   /* *preprocessing* */
**for** $i = 1$ **to** $n$ **do**
    /* *assume* $X_i = X_l \cdot X_r$ *for* $l, r < i$ */
      $pos := Solution(|P|, OV[X_l, P], OV[P, X_r])$;
      **if** $pos \neq 0$ **then** report an occurrence and STOP
```

THEOREM 5. *The algorithm Pattern_Matching works in $O(n^4 \log n)$ time.*

PROOF. First we show that *Solution*$(|P|, OV[X_i, P], OV[P, X_j])$ is performed by at most $2n$ applications of this operation *Solution*$(|P|, U, V)$, where both U and V are single arithmetic progressions. Remember that the set $OV[X_i, P]$ is expressed by at most $k(\le n)$ segments $U_1, U_2, \ldots, U_k$ in a sorted order, where each U_i is a single arithmetic progression. The same thing can be said for $OV[P, X_j]$ by the segments $W_1, W_2, \ldots, W_l$ $(l \leq n)$. In order to compute *Solution*$(|P|, OV[X_i, P], OV[P, X_j])$, we have only to perform the operation *Solution*$(|P|, U_i, V_j)$ for all pairs of U_i and V_j such that $\min(U_i \oplus V_j) \leq |P| \leq \max(U_i \oplus V_j)$. It is not hard to verify that the number of such pairs is at most $k + l \leq 2n$, and we can enumerate all such pairs in $O(n)$ time, since the segments are sorted. Since each operation *Solution*$(|P|, U_i, V_j)$ can be done in $O(n^2)$ time by Lemma 5, the operation *Solution*$(|P|, OV[X_i, P], OV[P, X_j])$ runs in $O(n^3)$ time. The algorithm *Pattern_Matching* makes $O(n)$ such operations, hence the total complexity for all these operations is $O(n^4)$. By Theorem 4, the compressed overlap table can be constructed in $O(n^4 \log n)$ time, so the whole algorithm *Pattern_Matching* works in asymptotically the same time. This completes the proof of this theorem (and also of our main result: Theorem 1). □

5. An NP-Complete Version of Pattern-Matching

Let *Var* be a set of variables in some straight-line program of length n over an alphabet Σ, and P be a pattern (given by a straight-line program $\mathcal{P}$ of length $m \leq n$). For a regular expression W, let $L(W)$ be the language defined by W. 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 $P \in \nu(W)$,

where $\nu(W) \subseteq \Sigma^*$ is the language obtained from $L(W)$ by expanding the values of variables in *Var*.

THEOREM 6. *The regular-expression-matching problem for shortly described strings is NP-complete, even if the expressions are $*$-free and contain no empty string 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 Karp [1972], and Garey and Johnson [1979], pp. 223. We can construct easily a straight-line program such that the value of X_i is 1^{a_i} and $P = 1^K$, where $\Sigma = \{1\}$. Then the SUBSET SUM problem is reduced to the membership:

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

The empty string ε can be easily eliminated by rescaling numbers and replacing ε by a single letter $\mathtt{1}$. This completes the proof. □

References

CROCHEMORE, M. AND RYTTER, W. 1994. *Text Algorithms.* Oxford University Press, New York.

FARACH, M. AND THORUP, M. 1995. String-matching in Lempel-Ziv compressed strings. In *27th ACM STOC*, 703–713.

GAREY, M.R. AND JOHNSON, D.S. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman.

KARP, R.M. 1972. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, Miller, R.E. and Thatcher, J.W., Editors. Plemum Press, New York, 85–103.

KARPINSKI, M., RYTTER, W., AND SHINOHARA, A. 1995. Pattern-matching for strings with short descriptions. In *Proc. Combinatorial Pattern Matching*, Volume 637 of *Lecture Notes in Computer Science*. Springer-Verlag, 205–214.

KNUTH, D. 1981. *The Art of Computing Vol. II: Seminumerical Algorithms. Second edition.* Addison-Wesley.

LOTHAIRE, M. 1983. *Combinatorics on Words.* Addison-Wesley.

PLANDOWSKI, W. 1994. Testing equivalence of morphisms on context-free languages. In *ESA'94*, Volume 855 of *Lecture Notes in Computer Science*. Springer-Verlag, 460–470.