An Improved Pattern-Matching Algorithm for Strings with Short Descriptions

Marek Karpinski[1] * **Wojciech Rytter**[1,2] † **Ayumi Shinohara**[1,3]

[1] Department of Computer Science, University of Bonn
53117 Bonn, Germany
marek@cs.uni-bonn.de

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

[3] Research Institute of Fundamental Information Science,
Kyushu University 33, Fukuoka 812, Japan
ayumi@rifis.kyushu-u.ac.jp

Abstract

We improve the time complexity of the pattern matching problem for strings which are succinctly described in terms of straight-line programs (or alternatively in terms of context-free grammars or recurrences). Examples of such strings are *Fibonacci words* and *Thue-Morse words*, see [4]. Usually the strings of descriptive size n are of exponential length with respect to n. A complicated algorithm for the *equality-test* (testing if two shortly described strings are the same) in $O(n^4)$ time was constructed in [6]. This algorithm was extended in [5] to the *pattern-matching* problem by using $O(n^3)$ instances of the *equality-test*, this gave $O(n^7)$ time. In this paper we reduce the time complexity to $O(n^4 \log n)$. We show that the pattern matching for shortly described strings can be done without applying an algorithm from [6] and the problem has the similar asymptotic complexity as the best algorithm for the equality-test. The latter problem is a special instance of the pattern-matching problem and can be solved by our algorithm. The crucial point in the algorithm is the succinct (linear size) representation of all (potentially many) periods of a string of an exponential size. The structure of our algorithm is *bottom-up*, while the construction of [6] is quite different and works *top-down* in the sense of evaluation trees for straight-line programs (or derivation trees in the case of grammars).

*Research partially supported by the DFG Grant KA 673/4-1, and by the ESPRIT BR Grants 7097 and ECUS 030.
†Supported by the DFG grant.

1 Introduction

The *pattern-matching* problem is the central algorithmic problem related to *texts*, see [1]. Recently the problem was considered in a *compressed setting*, see [2]. We use a simpler (compared with [2]) type of compression in terms of straight-line programs (grammars, or recurrences). Examples of typical words described in this way are *Fibonacci words* and *Thue-Morse* words. Such type of a short description (compression) is *enough powerful* to describe very *succinctly* some interesting words, and simultaneously it is *enough simple* to achieve the time complexity of the pattern-matching problem similar to the complexity of the equality-testing. Our *main aim* is to extend the result of [6] from the equality-testing problem to the pattern-matching problem. The main difference between our results and the results in [2] is that in [2] patterns are assumed to have *explicit* representations, while we allow patterns to be given *implicitly* by a short description, so our results and results in [2] are rather incomparable.

Our algorithm works for an *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 $N = \Omega(2^{c \cdot n})$.

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 $n = |\mathcal{R}|$.

For a string w denote by $w[i..j]$ the subword of w starting at i and ending at j. Similarly for a variable X denote $X[i..j] = \nu(X)[i..j]$. We can later identify variables with their values.

Denote by $\mathcal{P}$ and $\mathcal{T}$ the descriptions of a 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 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 [4].

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

We can see that

$$\begin{array}{rclcrcl} T & = & \nu(\mathcal{T}) & = & \nu(X_8) & = & \mathtt{abaababaabaababaababa}, \\ P & = & \nu(\mathcal{P}) & = & \nu(Y_7) & = & \mathtt{aabaababa} \end{array}$$

as shown in Figure 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.*

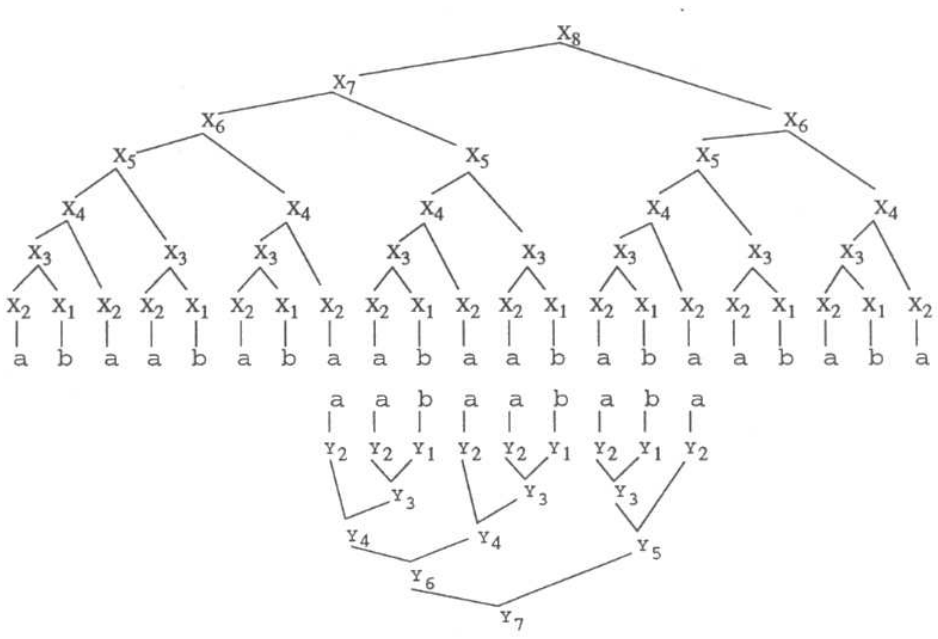

Figure 1: The evaluation trees for the text T (top-down) and the pattern P (bottom-up), there is an occurrence of the pattern $\mathcal{P}$ starting at position 8 of $\mathcal{T}$.

2 The succinct representation of the overlap structure

An *overlap* is a triple $Overlap(X, Y, k)$, where $k \leq |X|, |Y|$. We say that this overlap is *valid* iff $Y[1..k]$ is a suffix of X, see an example in Figure 2.

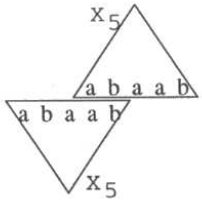

Figure 2: $Overlap(X_5, X_5, 2)$: an example of an overlap for the variables in the straight-line program $\mathcal{T}$ from Example 1.

The *overlap query* is the question of the type: is $Overlap(X, Y, k)$ valid?

The *overlap structure* of a straight-line program $\mathcal{R}$ represents all (potentially exponentially many) valid overlaps between variables of $\mathcal{R}$ and allows to answer each *overlap query* in short time. We show that using the periodicity approach a *succinct representation* is possible using only $O(n^3)$ space.

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

	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 1: The overlap structure of the example straight-line program $\mathcal{T}$. For each X_i and X_j, the content at column X_i and row X_j is the set of all k's such that $Overlap(X_i, X_j, k)$ is valid.

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 iff $w[i] = w[i-p]$, whenever both

sides are defined. Hence $p = |w|$ and $p = 0$ are considered to be periods.

Lemma 1 (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. The following fact is a consequence of Lemma 1.

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

Lemma 2 (*applying periodicity 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 1 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 by inductive assumption, since $\lfloor \log_2(|w| - q) \rfloor < j$ and A is a single progression. ∎

The description of the succinct representation of the overlap structure

The set of all overlaps of variables of a given straight-line program $\mathcal{R}$ is represented by the table $OV[X_i, X_j]$, $1 \leq i, j \leq n$, where:

$$OV[X_i, X_j] = \{k \ : \ Overlap(X_i, X_j, k) \text{ is valid } \} \ .$$

The value of $OV[X_i, X_j]$ is stored as a pair $(k, Periods(X_j[1..k]))$, where $k = \max\{p \ : \ p \in OV[X_i, X_j]\}$. The set $Periods(X_j[1..k])$ is stored in a succinct way. According to Lemma 2 this set consists of $O(n)$ arithmetic progressions.

So we maintain the value of each such set as a description of a linear number of progressions, for each progression only the first and the second element is needed (in this particular case each progression continues until it is outside k). Assume that these sets are sorted with respect to the first element of each progression.

The following lemma follows directly from the fact that we keep sets the progressions (related to overlaps) in a sorted order.

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

The description of the operation *Compress*

Assume we have a (possibly redundant) representation V of the set of all overlaps between two variables X_i, X_j in terms of a set V of progressions corresponding to periods of $Periods(X_j[1..k])$ for some k. 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, having representations (of a constant size) for each progression. The resulting set is denoted by $Compress(V)$.

3 The First Mismatch and the Period Continuation

Assume $\mathcal{R}$ is a given straight-line program, whose variables are $X_1, X_2, \ldots, X_n$. If $i < j$ then we say that X_i precedes X_j.
Assume $|Y| \geq k$. Define $FirstMismatch(X, Y, k)$ as a first mismatch (from left) which is a witness to the fact that $Overlap(X, Y, k)$ is not valid, more formally:

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

If there is no such i then the value of $FirstMismatch(X, Y, k)$ equals *nil*.

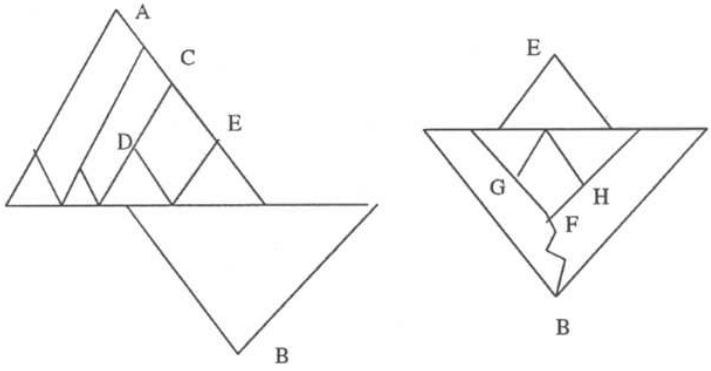

Figure 3: Looking for the first mismatch in $Overlap(A, B, k)$.

Theorem 2 *Assume A and B are two variables of a given straight-line program. 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, B. The internal nodes can be identified with corresponding variables. We use a kind of *binary 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

Figure 3. 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 Figure 3 (on the left).

Then we check if the overlap between B and D is valid.

If "no" then we search recursively for a mismatch in the overlap between D and B.

If "yes" then 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 B. Then we test the validity of the overlap 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 linear. This completes the proof. ∎

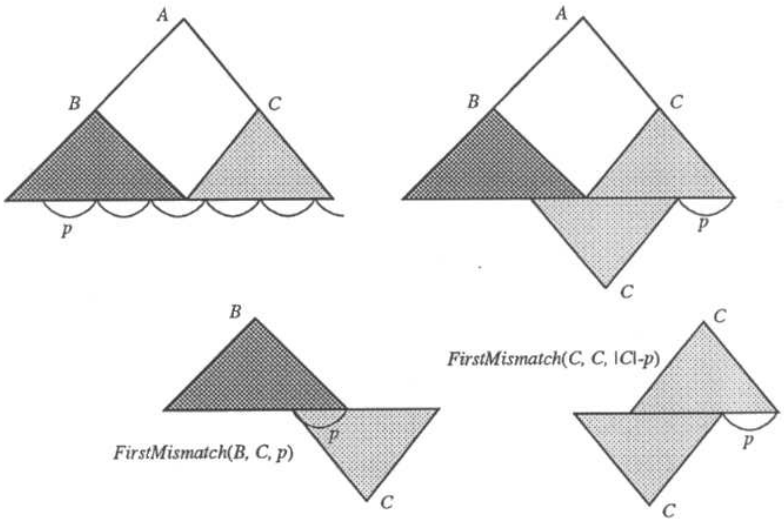

Figure 4: We know that in A there is the period p in the part B. The first mismatch to the continuation of this periodicity in C is found by two applications of searching first mismatches in overlaps.

Theorem 3 *Assume the succinct representation of the overlap structure of all variables preceding A 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 Figure 4. 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 $FirstMismatch(C, C, |C| - p)$. ∎

4 Preprocessing: computing the overlap structure 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 (succinctly represented) overlap structure 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 Figure 5. The basic auxiliary operation is the *prefix extension*.

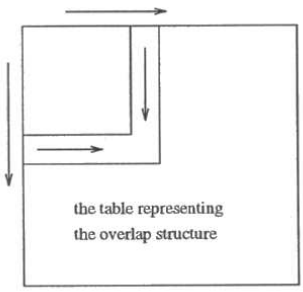

Figure 5: The order of processing elements of the table OV.

Assume that $U \subseteq [1..n]$. 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:

Observation

Assume that $X_i = X_p \cdot X_q$ and $U := OV[X_p, X_j]$ and $W := OV[X_j, X_q]$. Then

$$OV[X_i, X_j] := Compress(PrefExt(U, X_j, X_q) \cup W).$$

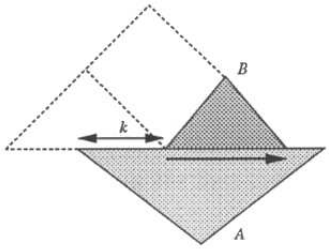

Figure 6: $k + |B| \in PrefExt(U, A, B)$ iff the common parts of A and B agree.

```
ALGORITHM Compute_Overlap_Structure ;
    compute OV[X_i, X_j] for each pair of terminal variables;
    for (i, j) in the order shown in Figure 5 do
        begin
            {Assume X_i = X_p · X_q}
            { computation of OV[X_i, X_j] }
            U := OV[X_p, X_j];
            U := PrefExt(U, X_j, X_q);
            W := OV[X_j, X_q];
            OV[X_i, X_j] := Compress(U ∪ W);
        end
```

The next lemma says that the operations *PrefExt* is enough efficient in case of a single progression. The lemma was essentially proved in [5], we have only to observe additionally (compared with proof in [5]) that the period continuation can be computed in $O(n \log n)$ time.

Lemma 4

Assume A and B are two variables and the overlap structure for variables which precede A or B is computed. Let $S = \{t_0, t_1, \ldots, t_s\} \subseteq [1 \ldots k]$ be a linear set given by its succinct representation, where $t_0 = k$ and strings $x_i = A[1 \ldots t_i]$, $0 \leq i \leq s$, are suffixes of $A[1 \ldots k]$. 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 Figure 6. 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. The 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 no 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 Figure 7 (case A).

Case B: there is a mismatch in Z and a mismatch in 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 7 (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.

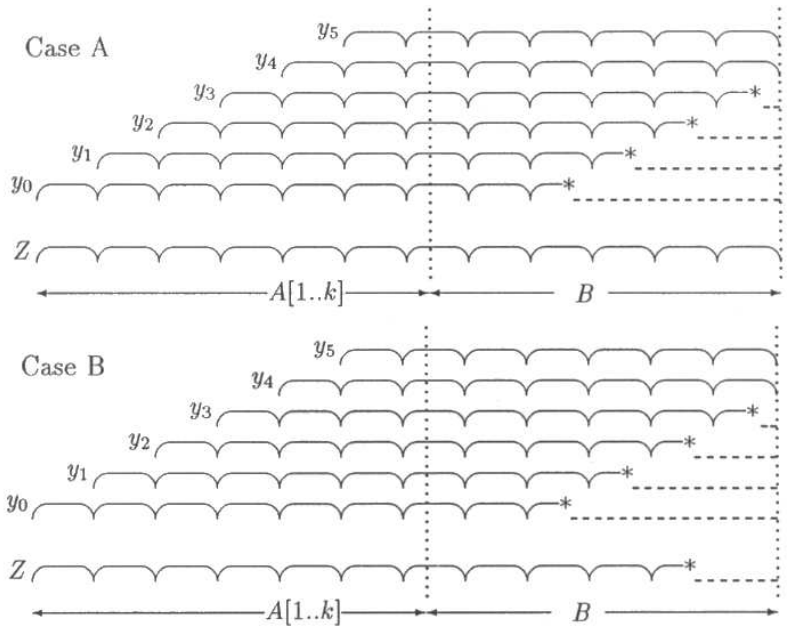

Figure 7: Two cases: $Z = A[1..k] \cdot B$ has (has no) mismatch, $y_i = A[1..|X_i| + |B|]$.

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 $\{|y_i| : i \text{ is a good index } \}$ is linear. This completes the proof. ∎

The set U in the algorithm consists of a linear number of 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 succinct representation of the overlap structure 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.

Theorem 5 *Assume $\mathcal{X}$ is a string given by its description of size n. Then we can compute in $O(n^4 \log(n))$ time a linear size representation of the set Periods$(\nu(\mathcal{X}))$.*

5 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.

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 (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 $O(n^2)$ 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
>
> $$ax + by = c, \ \ (1 \le x \le A, 1 \le y \le B). \qquad (1)$$

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 *hitten* by this line. This can be done in $O(n^2)$ time using operations *div* and *mod* on integers. We refer for details to [3] (see page 325 and Exercise 14 on page 327). ∎

ALGORITHM *PATTERN_MATCHING* ;
Compute_Overlap_Structure ; { preprocessing }
for $k = 1$ **to** n **do**
  {assume $X_k = X_i \cdot X_j$ for $i,j < k$ }
    $pos := Solution(|P|, OV[X_i, P], OV[P, X_j])$;
    if $pos \neq 0$ **then** report an occurrence and STOP

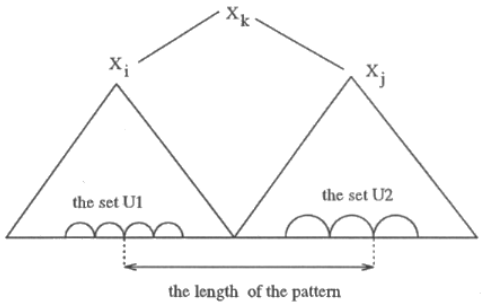

Figure 8: $U1 = OV[X_i, P]$ and $U2 = OV[P, X_j]$, the algorithm find positions in these sets whose difference is the length of the pattern.

Theorem 6 *The algorithm PATTERN_MATCHING works in $O(n^4 \log n)$ time.*

Proof. It can be shown that the operation $Solution(|P|, OV[X_i, P], OV[P, X_j])$ can be done by applying $O(n)$ applications of this operation $Solution(c, U, W)$ for sets U, W which are arithmetic progressions, each of these operations can be done in $O(n^2)$ time. Hence the operation $Solution(|P|, OV[X_i, P], OV[P, X_j])$ can be performed in $O(n^3)$ time. The algorithm makes $O(n)$ such operations, hence the total complexity for all these operations is $O(n^4)$. The overlap structure 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). ∎

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

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

[5] M. Karpinski, W. Rytter, A. Shinohara, "Pattern-matching for strings with short descriptions", to appear in the proceedings of Combinatorial Pattern Matching 1995, preliminary version in Research Report, Institut für Informatik der Universität Bonn, No. 85124-CS (1995)

[6] W. Plandowski, "Testing equivalence of morphisms on context-free languages", ESA'94, Lecture Notes in Computer Science 855, Springer-Verlag, 460–470 (1994).