Testing string superprimitivity in parallel

Dany Breslauer

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

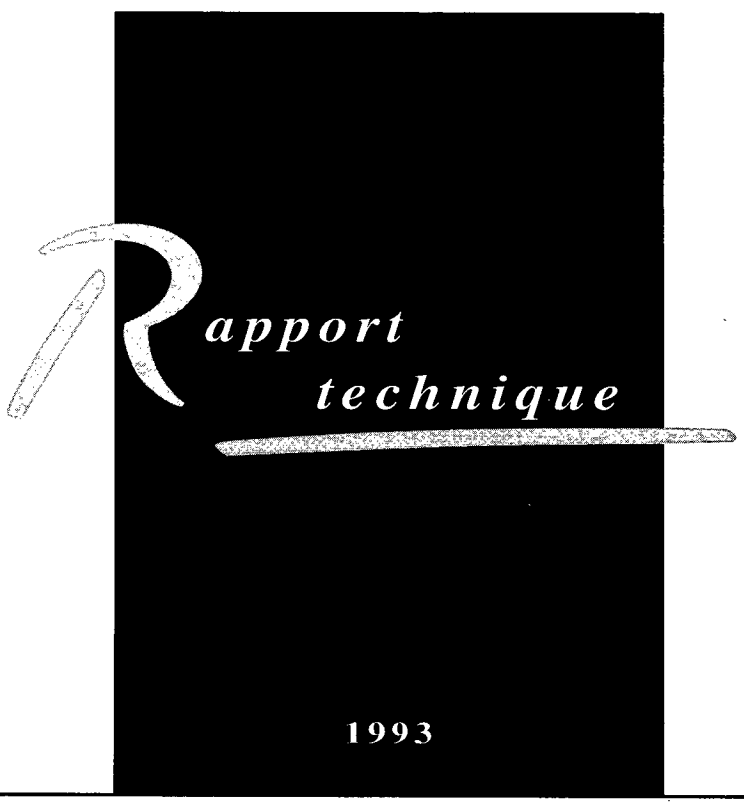
Testing String Superprimitivity in Parallel

Dany BRESLAUER

N° 160
Septembre 1993

PROGRAMME 2

Calcul symbolique,
programmation
et génie logiciel



Testing String Superprimitivity in Parallel

Dany Breslauer*
Institut National de Recherche en Informatique
et en Automatique
B.P. 105, 78153 Le Chesnay Cedex, France

Abstract

A string w *covers* another string z if every symbol of z is within some occurrence of w in z. A string is called *superprimitive* if it is covered only by itself, and *quasiperiodic* if it is covered by some shorter string. This paper presents an optimal $O(\alpha(|z|)\log\log|z|)$ time CRCW-PRAM algorithm that tests if a string z is superprimitive, where $\alpha(|z|)$ is the inverse of the Ackermann function. An alternative implementation takes $O(\log\log|z|)$ time using $\frac{|z|\log|z|}{\log\log|z|}$ processors, and is the fastest possible with this number of processors over a general alphabet.

Test de la superprimitivité d'une chaine en parallèle.

Résumé

Une chaine w *recouvre* une autre chaine z si chaque symbole de z apparait dans une occurrence de w dans z. Une chaine est dite *superprimitive* si elle n'est recouverte que par elle-même, et *quasi-périodique* si elle est recouverte par une chaine plus courte. Ce papier presente un algorithme CRCW-PRAM de test de superprimitivité optimal, de complexité en temps $O(\alpha(|z|)\log\log|z|)$, où $\alpha(|z|)$ est l'inverse de la fonction d'Ackermann. Une deuxième implémentation réalise une complexité en temps de $O(\log\log|z|)$ avec $\frac{|z|\log|z|}{\log\log|z|}$ processeurs. Elle est la plus rapide possible sur un alphabet général avec ce nombre de processeurs.

*The author was partially supported by the European Research Consortium for Informatics and Mathematics postdoctoral fellowship. Part of this work was done while visiting at the Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands.

1 Introduction

Quasiperiodicity, as defined by Apostolico and Ehrenfeucht [2], is a regularity of strings that is closely related to other regularities such as periods and squares [14]. Apostolico, Farach and Iliopoulos [3] and Breslauer [6] gave linear-time sequential algorithms that tests if a string is superprimitive. Apostolico and Ehrenfeucht [2] presented an algorithm that finds all maximal quasiperiodic substrings of a string. Iliopoulos et al. [13] discuss a similar notion of string covering.

A parallel algorithm is said to be *optimal*, or to achieve *an optimal speedup*, if its time-processor product is the same as the running time of the fastest sequential algorithm for the same problem. This paper presents an optimal $O(\alpha(|z|)\log\log|z|)$ time $\frac{|z|}{\alpha(|z|)\log\log|z|}$-processor CRCW-PRAM algorithm that tests if a string z is superprimitive. An alternative implementation takes $O(\log\log|z|)$ time using $\frac{|z|\log|z|}{\log\log|z|}$ processors. Both algorithms work under the general alphabet assumption where the only access they have to the input string is by pairwise symbol comparisons. We show by a reduction to a known lower bound for string matching that the latter algorithm is the fastest possible with the number of processors used. Note that there exists a trivial constant time superprimitivity testing algorithm that uses n^2 processors.

The superprimitivity testing algorithm follows techniques that were used in solving several other parallel string problems [1, 7]. In particular, it uses the parallel string matching algorithm of Breslauer and Galil [8] as a procedure that solves several string matching problems simultaneously and then combines the results of the string matching problems into an answer to the superprimitivity problem.

The paper is organized as follow. Section 2 gives basic definitions and properties of strings. Section 3 overviews the parallel algorithms and tools that are used in the superprimitivity testing algorithm. Section 4 describes the basic step which is used by the superprimitivity testing algorithm in Section 5. The lower bound is given in Section 6 and some concluding remarks are given in Section 7.

2 Properties of Strings

A string w *covers* a string z if for every position $i \in \{1, \cdots, |z|\}$ of z there exists an occurrence of w starting at some position j of z, such that $1 \leq j \leq i \leq j + |w| - 1 \leq |z|$. A string z is called *superprimitive* if it is covered only by itself, and *quasiperiodic* if it is covered by a string w, such that $w \neq z$. A superprimitive string w that covers a string z is called a *quasiperiod* of z.

A string z has a *period* w if z is a prefix of w^k for some integer k. The shortest period of a string z is called *the period* of z. Clearly, a string is always a period of itself.

We say that a non-empty string w is a *border* of a string z if z starts and ends with an occurrence of w. That is, $z = uw$ and $z = wv$ for some possibly empty strings u and v. Clearly, a string is always a border of itself. This border is called the trivial border.

We give next few simple facts which are used in the superprimitivity testing algorithm. The first four facts were used in the sequential algorithms [3, 6] where their proofs can be found.

Fact 2.1 *A string z has a period of length π, such that $\pi < |z|$, if and only if it has a non-trivial border of length $|z| - \pi$.*

Fact 2.2 *If a string w covers a string z then w is a border of z.*

Note that by the last fact any cover of a string z can be represented by a single integer that is the length of the border of z.

Fact 2.3 *If a string w covers a string z and another string v, such that $|w| \leq |v|$, is a border of z then w covers also v.*

Fact 2.4 *If a string z is covered by two strings w and v, such that $|w| \leq |v|$, then w covers v. Therefore, a string cannot have two different quasiperiods.*

Fact 2.5 *If a string z has a border w, such that $2|w| \geq |z|$, then w covers z.*

Proof: w covers the first half of z since it is a prefix of z and the last half of z since it is also a suffix. Therefore, all symbols of z are covered by w. □

3 The CRCW-PRAM Model

The algorithms described in this paper are for the concurrent-read concurrent-write parallel random access machine model. We use the weakest version of this model called the *common CRCW-PRAM*. In this model many processors have access to a shared memory. Concurrent read and write operations are allowed at all memory locations. If several processors attempt to write simultaneously to the same memory location, it is assumed that they write the same value. The superprimitivity testing algorithm uses the following previously known algorithms:

1. The parallel string matching algorithm of Breslauer and Galil [8] that finds all occurrences of a pattern string of length m in a text string of length n in $O(\log\log m)$ time using $\frac{n}{\log\log m}$ processors. By a lower bound of Breslauer and Galil [9], this algorithm is the fastest possible optimal parallel string matching algorithm on a general alphabet.

2. The parallel algorithm of Breslauer and Galil [1, 10] that finds all periods of a string of length n in $O(\log\log n)$ time using $\frac{n}{\log\log n}$ processors.

3. The algorithm of Fich, Ragde and Wigderson [11] to compute the minimum of n integers between 1 and n in constant time using n processors.

One of the major issues in the design of PRAM algorithms is the assignment of processors to their tasks. We ignore this issue and use a general theorem that states that the assignment can be done.

Theorem 3.1 *(Brent [5]) Any synchronous parallel algorithm of time t that consists of a total of x elementary operations can be implemented on p processors in $\lceil x/p \rceil + t$ time.*

4 The Basic Step

This section shows how to test efficiently whether a string w covers another string z.

Lemma 4.1 *Given two strings z and w, there exists an algorithm that tests whether w covers z in $O(\log\log|z|)$ time and using $O(|z|)$ operations.*

Proof: Using Breslauer and Galil's string matching algorithm, find all occurrences of w in z. This computation takes $O(\log\log|z|)$ time and uses $O(|z|)$ operations. It remains to check if each symbol of z is covered by some occurrence of w.

Partition the string z into $B = \lfloor |z|/|w| \rfloor$ consecutive blocks of length $|w|$, ignoring the last block if it is shorter. In each block simultaneously, find the indices of the beginning of the first and last occurrences of w in block number i and call these indices f_i and l_i, respectively. This computation can be done by using Fich, Ragde and Wigderson's integer minima algorithm in constant time and using $O(|z|)$ operations.

If there are no occurrences of w in one of the blocks, then clearly some symbols in that block are not covered by an occurrence of w. Otherwise, l_i and f_i are defined for each block. In each block, all symbols at positions between f_i and l_i are obviously covered. The symbols between the last occurrence of w in a block and the first occurrence of w in the following block are covered if and only if $f_{i+1} \leq l_i + |w|$. Special attention is required in the first and the last blocks, where it is necessary to check if the beginning and the end of z are also covered by testing if $f_1 = 1$ and if $l_B = |z| - |w| + 1$. These conditions can be checked simultaneously in each block in constant time and using a constant number of operations per block. Thus, the number of operations used is $O(|z|)$ and the time is $O(\log\log|z|)$. □

5 The Superprimitivity Test

This section describes the superprimitivity testing algorithm. The algorithm finds the quasiperiod of a string z by starting with all borders of z as candidates for the quasiperiod, and continues eliminating candidates until the quasiperiod is found. Clearly, z is superprimitive if and only if it is the quasiperiod of itself.

To obtain an efficient algorithm we try to have a small number of candidates to start with. The following lemma shows how this can be achieved.

Lemma 5.1 *It is possible to reduce the number of candidates for the quasiperiods of z to $\lceil \log|z| \rceil$ in $O(\log\log|z|)$ time and using $O(|z|)$ operations.*

Proof: Find all borders of z using Breslauer and Galil's algorithm that finds all periods of a string. Recall that by Fact 2.2, if w covers z then w must be a border of z and by Fact 2.1, there is a one-to-one correspondence between the borders and the periods of a string.

Partition the borders of z into $\lceil \log|z| \rceil$ intervals $[2^i \cdots 2^{i+1} - 1]$ according to their length. If there are few borders whose length is in the same interval, then by Fact 2.5, the shortest border covers the longer ones. By Fact 2.4, only the shortest border in each interval is a candidate for the quasiperiod of z. The shortest border in each interval can be found in constant time and using $O(|z|)$ operations by Fich, Ragde and Wigderson's integer minima algorithm, in all intervals simultaneously.

Thus, the number of candidates for the quasiperiod of z was reduced to at most $\lceil \log|z| \rceil$ candidates in $O(\log\log|z|)$ time and using $O(|z|)$ operations. Furthermore, these candidates satisfy that there are at most i candidates whose length is larger than or equal to $|z|/2^{i-1}$. □

We devise a sophisticated approach to eliminate candidates for the quasiperiod. It consists of a repeated application of the following lemma.

Lemma 5.2 *Let $b_0, \cdots, b_q$, such that $|b_0| < \cdots < |b_q|$, be borders of z that are candidates for the quasiperiod of z. Then in $O(\log\log|b_q|)$ time and using $O(q|b_q|)$ operations, it is possible to eliminate all but one of these borders.*

Proof: Using the algorithm in Lemma 4.1, check simultaneously if each of the candidates b_i covers the longest candidate b_q. If b_i is the quasiperiod of z, then by Fact 2.3, b_i covers b_q. By Fact 2.3, if b_i covers b_q, then b_i also covers b_j, for $j = i, \cdots, q$. Therefore, the shortest border b_i that covers b_q is the only candidate to be the quasiperiod of z.

The q covering tests by Lemma 4.1 take $O(\log\log|b_q|)$ time and use $O(q|b_q|)$ operations. The shortest border is found by Fich, Ragde and Wigderson's integer minima algorithm in constant time and $O(q)$ operations. □

After the initial elimination of candidates in Lemma 5.1 we are left with at most $\lceil\log|z|\rceil$ candidates for the quasiperiod and continue to eliminate more candidates. The simplest approach is the following.

Lemma 5.3 *There exists an algorithm that finds the quasiperiod of a string z in $O(\log\log|z|)$ time using $\frac{|z|\log|z|}{\log\log|z|}$ processors.*

Proof: Let b_i, $i = 1, \cdots, \lceil\log|z|\rceil$, be the borders of z that survive after the initial elimination of candidates in Lemma 5.1. By Lemma 5.2, all but one of these candidates can be eliminated in $O(\log\log|z|)$ time and using $O(|z|\log|z|)$ operations. The remaining candidate is the quasiperiod of z. By Theorem 3.1, the whole algorithms takes $O(\log\log|z|)$ time using $\frac{|z|\log|z|}{\log\log|z|}$ processors. □

We refine the method above to obtain a fast optimal algorithm.

Theorem 5.4 *There exists an algorithm that finds the quasiperiod of a string z in $O(\alpha(|z|)\log\log|z|)$ time using $\frac{|z|}{\alpha(|z|)\log\log|z|}$ processors.*

Proof: The algorithm starts with the initial elimination of candidates in Lemma 5.1, after which it is left with at most $\lceil\log|z|\rceil$ borders that are candidates for the quasiperiod. The rest of the algorithm consists of several iteration in which candidates for the quasiperiod of z are eliminated using Lemma 5.2.

The algorithm partitions the candidates remaining after iteration number j into groups, and uses Lemma 5.2, in all groups simultaneously, to eliminate all but one candidate in each group. The groups are carefully chosen in such a way that the total number of operations used in all groups throughout all iterations is $O(|z|)$, and thus the algorithm is optimal.

In the description below, we use only upper bounds on the lengths of the surviving candidates. It is understood that each length bound has a candidate that it is associated with. Sometime, the initial number of candidates is smaller, but we rather ignore this possibility for simplicity of our notation (one can assume that there are many candidates whose length is 0).

Define $Q_0(i) = 2^i$ and $I_j(n) = \min\{i | Q_j(i) \geq n\}$. We maintain that after j iterations, the number of remaining candidates for the quasiperiod is at most $I_j(|z|)$, in addition to j candidates that are handled separately. Furthermore, if the $I_j(|z|)$ candidates are ordered by decreasing lengths and numbered starting from 0, then the length of candidate number i is at most $|z|/Q_j(i)$. By Lemma 5.1, $I_0(|z|) = \lceil\log|z|\rceil$ and the lengths of the initial candidates are smaller then $|z|/Q_0(i)$, for $i = 0, \cdots, I_0(|z|)$.

We maintain that the number of operations made in group number i in iteration number j is at most $O(|z|/2^{i+j})$. Thus, the total number of operation made in all groups in iteration number j is at most $O(|z|/2^j)$ and the number of operations in all iterations is $O(|z|)$. In order to achieve this, we do not consider the longest candidate surviving iteration number $j-1$ in the next iterations and deal separately with the left-out candidates later. Note that the length of this candidate can be as large as $|z|/Q_{j-1}(0)$. Thus, the length of the longest candidate from iteration number $j-1$ that is considered in iteration j is at most $|z|/Q_j(0)$, for $Q_j(0) = Q_{j-1}(1)$.

The surviving candidates of iteration $j-1$ are partition into groups. If the length of the longest member in group number i is at most $n/Q_j(i)$, then we choose the size of the group to be $1 + Q_j(i)/2^{i+j}$. The number of operations used in eliminating all but one of the candidates in the group using Lemma 5.2 is $O(|z|/2^{i+j})$ as required.

We partition the candidates surviving iteration $j-1$ into groups by this rule. Therefore, the first i groups in iteration j include the longest $i + \Sigma_{k=0}^{i-1} Q_j(k)/2^{k+j}$ candidates surviving iteration $j-1$, apart from the longest candidate that is left out. The length of the longest candidate in group number $i+1$ is bounded by $|z|/Q_j(i)$, where

$$Q_j(i) = Q_{j-1}(i + 1 + \Sigma_{k=0}^{i-1} Q_j(k)/2^{k+j}).$$

The iterations terminate when there is only one candidate left. Namely, after $\hat{\alpha}(|z|)$ iterations, where $\hat{\alpha}(n) = \min\{j | Q_j(1) \geq n\}$. Note that there are $\hat{\alpha}(|z|)$ left-out candidates that were not eliminated yet. It is possible to eliminate all but one of these candidates and to find the quasiperiod of z by applying Lemma 4.1 $\hat{\alpha}(|z|)$ times. (This is done by considering the shortest two remaining candidates, checking if the shorter covers the longer and eliminating one of the two by Lemma 2.3. There are clearly $\hat{\alpha}(|z|)$ such steps until one candidate remains and the number of operations used is $O(|z|)$.)

The whole algorithm uses $O(|z|)$ operations and takes $O(\hat{\alpha}(|z|) \log\log|z|)$ time. By Theorem 3.1, it can be implemented optimally using $\frac{|z|}{\hat{\alpha}(|z|)\log\log|z|)}$ processors.

The bounds above involve an extremely slow growing function $\hat{\alpha}(n)$ that is a close relative of the inverse Ackermann function. The inverse Ackermann function appears in the running time of several sequential and parallel algorithms [4, 12, 15, 16]. Ackermann's function is defined as [16]: $A_1(n) = 2^n$, for $n \geq 1$, $A_k(1) = A_{k-1}(2)$, for $k \geq 1$, and $A_k(n) = A_{k-1}(A_k(n-1))$, for $k, n \geq 2$. The inverse-Ackermann function (with one parameter) is defined as: $\alpha(n) = \min\{k | A_k(1) \geq n\}$. It is not difficult to verify that $A_k(n) \leq Q_k(n)$ and $\hat{\alpha}(n) \leq \alpha(n)$. □

In fact, one can generalize the algorithms in Lemma 5.3 and in Theorem 5.4 and obtain an non-optimal $O(\log\log|z|)$ time algorithm that uses few processors.

Corollary 5.5 *There exists an algorithm that finds the quasiperiod of a string z in $O(\log\log|z|)$ time using $\frac{|z|I_c(|z|)}{\log\log|z|}$ processors, for an integer constant $c \geq 0$.*

Proof: Let the algorithm in Theorem 5.4 eliminate candidates for the quasiperiod for c iterations, after which there are $I_c(|z|)+c$ candidates left. This takes $O(\log\log|z|)$ time and $O(|z|)$ operations.

Then, continue and eliminate all but one of the remaining candidates, simultaneously, using Lemma 5.2. The whole algorithm takes $O(\log\log|z|)$ time and uses $O(|z|I_c(|z|))$ operations. By Theorem 3.1, it can be implemented within the claimed bounds. Note that the algorithm in Lemma 5.3 is a special case of this algorithm when $c = 0$. □

6 The Lower Bound

We prove a lower bound for testing if a string is superprimitive by a reduction to the lower bound for string matching by Breslauer and Galil [9]. That lower bound is on the number of comparison rounds performed by an algorithm that computes the period length of a string. The lower bound holds for the CRCW-PRAM model in case of a general alphabet where the only access an algorithm has to the input strings is by pairwise comparisons of symbols.

Breslauer and Galil [9] show that an adversary can fool any algorithm which claims to test if a string of length n has a period that is shorter than half of its length in less than $\Omega(\lceil \frac{n}{p} \rceil + \log\log_{\lceil 1+p/n \rceil} 2p)$ rounds of p comparisons each. Without going into the details of that lower bound, we use the fact that the adversary answers comparisons in such a way that after $\Omega(\lceil \frac{n}{p} \rceil + \log\log_{\lceil 1+p/n \rceil} 2p)$ rounds it is still possible that the input string has a period that is shorter than half of its length or that is does not have any such period. In the latter case there is at least one symbol of the string that does not occur anywhere else in the string.

Lemma 6.1 *The string generated by Breslauer and Galil's adversary is superprimitive if and only if it does not have a period that is shorter than half of its length.*

Proof: If a string has a period that is shorter than half of its length, then by Fact 2.1 it has a border that is longer than half of its length and by Fact 2.5 is quasiperiodic. On the other hand, if there is a symbol that occurs only once in a string, then it is superprimitive. □

The last lemma establishes that the lower bound of Breslauer and Galil holds also for superprimitivity testing.

Theorem 6.2 *Any comparison based parallel string superprimitivity test with p comparisons in each round must take at least $\Omega(\lceil \frac{|z|}{p} \rceil + \log\log_{\lceil 1+p/|z| \rceil} 2p)$ rounds.*

Corollary 6.3 *The $O(\log\log|z|)$ time algorithms that were described in Lemma 5.3 and in Corollary 5.5 are the fastest possible with the number of processors used.*

7 Concluding Remarks

The superprimitivity testing algorithm given in Theorem 5.4 requires the computation of the $Q_j(i)$ function and its inverse. Note that for our purpose, it is possible to compute $Q_0(i)$, for $i = 0, \cdots, \lceil \log n \rceil$, in $O(\log\log n)$ time using $\log n$ processors and then to compute the rest of the $Q_j(i)$ values that are needed by the algorithm, and the inverse of $Q_j(i)$, using one processor within the claimed time bounds. Berkman and Vishkin [4] discuss fast and efficient parallel computation of similar functions.

The algorithms described in this paper are almost optimal. We conjecture that an optimal $O(\log\log n)$ time algorithm exists. It might be possible to design such an algorithm by finding a more efficient procedure that checks if several borders cover a string.

8 Acknowledgments

I thank Alberto Apostolico, Omer Berkman, Pino Italiano and Mireille Régnier for several discussions and help in obtaining some bibliographic references. I also thank Mireille Régnier for the French translation of the abstract.

References

[1] A. Apostolico, D. Breslauer, and Z. Galil. Optimal Parallel Algorithms for Periods, Palindromes and Squares. In *Proc. 19th International Colloquium on Automata, Languages, and Programming*, pages 296–307. Springer-Verlag, Berlin, Germany, 1992.

[2] A. Apostolico and A. Ehrenfeucht. Efficient Detection of Quasiperiodicities in Strings. Technical Report 90.5, The Leonadro Fibonacci Institute, Trento, Italy, 1990.

[3] A. Apostolico, M. Farach, and C. S. Iliopoulos. Optimal superprimitivity testing for strings. *Inform. Process. Lett.*, 39:17–20, 1991.

[4] O. Berkman and U. Vishkin. Recursive Star-Tree Parallel Data Structure. *SIAM J. Comput.*, 22(2):221–242, 1993.

[5] R. P. Brent. Evaluation of general arithmetic expressions. *J. Assoc. Comput. Mach.*, 21:201–206, 1974.

[6] D. Breslauer. An On-Line String Superprimitivity Test. *Inform. Process. Lett.*, 44(6):345–347, 1992.

[7] D. Breslauer. Fast Parallel String Prefix-Matching. Technical Report CUCS-041-92, Computer Science Dept., Columbia University, 1992.

[8] D. Breslauer and Z. Galil. An optimal $O(\log \log n)$ time parallel string matching algorithm. *SIAM J. Comput.*, 19(6):1051–1058, 1990.

[9] D. Breslauer and Z. Galil. A Lower Bound for Parallel String Matching. *SIAM J. Comput.*, 21(5):856–862, 1992.

[10] D. Breslauer and Z. Galil. Finding all Periods and Initial Palindromes of a String in Parallel. Technical Report CUCS-017-92, Computer Science Dept., Columbia University, 1992.

[11] F. E. Fich, R. L. Ragde, and A. Wigderson. Relations between concurrent-write models of parallel computation. In *Proc. 3rd ACM Symp. on Principles of Distributed Computing*, pages 179–189, 1984.

[12] S. Hart and M. Sharir. Nonlinearity of Davenport-Schinzel sequences and generalized path compression schemes. *Combinatorica*, 6:151–177, 1986.

[13] C.S. Iliopoulos, D.W.G. Moore, and K. Park. Covering a String. In *Proc. 4rd Symp. on Combinatorial Pattern Matching*, 1993. To appear.

[14] M. Lothaire. *Combinatorics on Words.* Addison-Wesley, Reading, MA., U.S.A., 1983.

[15] R.E. Tarjan. Efficiency of a good but not linear set union algorithms. *J. Assoc. Comput. Mach.*, 22:215–225, 1975.

[16] R.E. Tarjan. *Data Structures and Network Algorithms.* SIAM, Philadelphia, PA., 1985.

Unité de Recherche INRIA Rocquencourt
Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 Le Chesnay Cedex (France)

Unité de Recherche INRIA Lorraine Technopôle de Nancy-Brabois - Campus Scientifique
615, rue du Jardin Botanique - B.P. 101 - 54602 Villers Lès Nancy Cedex (France)
Unité de Recherche INRIA Rennes IRISA, Campus Universitaire de Beaulieu 35042 Rennes Cedex (France)
Unité de Recherche INRIA Rhône-Alpes 46, avenue Félix Viallet - 38031 Grenoble Cedex (France)
Unité de Recherche INRIA Sophia Antipolis 2004, route des Lucioles - B.P. 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 Le Chesnay Cedex

ISSN 0249 - 0803