Two-Way String-Matching

MAXIME CROCHEMORE AND DOMINIQUE PERRIN

L.I.T.P., Institut Blaise Pascal, Université Paris 7, 2, Place Jussieu, Paris Cedex 05, France

Abstract. A new string-matching algorithm is presented, which can be viewed as an intermediate between the classical algorithms of Knuth, Morris, and Pratt on the one hand and Boyer and Moore, on the other hand. The algorithm is linear in time and uses constant space as the algorithm of Galil and Seiferas. It presents the advantage of being remarkably simple which consequently makes its analysis possible. The algorithm relies on a previously known result in combinatorics on words, called the *Critical Factorization Theorem*, which relates the global period of a word to its local repetitions of blocks.

Categories and Subject Descriptors: D.1.0 [**Programming Techniques**]: General; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems; G.2.2 [**Discrete Mathematics**]: Graph Theory; I.5 [**Computing Methodologies**]: Pattern Recognition; I.7 [**Computing Methodologies**]: Text Processing

General Terms: Algorithms, Design, Theory

Additional Key Words and Phrases: Analysis of Algorithms, combinatorial algorithms, pattern matching, text processing

1. *Introduction*

The problem of pattern recognition on strings of symbols has received considerable attention. In fact, most formal systems handling words can be considered as defining patterns in strings, as is the case for formal grammars and especially for regular expressions (see [31]) which provide a technique to specify simple patterns. Other kinds of patterns on words may also be defined (see for instance [1], [4], and [26]) but lead to less efficient algorithms.

The first version of this paper was presented at the Conference on Pattern Recognition held in Cefalù in September 1987 (CROCHEMORE, M., AND PERRIN, D. Pattern matching in strings. In DiGesu, ed., *Proceedings of the 4th Conference on Image Analysis and Processing*. Springer-Verlag, New York, 1988, pp. 67–79).

An improvement on this first version was presented at the Conference on the Foundations of Software Technology and Theoretical Computer Science held in Poona in December 1988 (CROCHEMORE, M. Constant-space string-matching. In Nori and Kumar, eds., *Foundations of Software Technology and Theoretical Computer Science*. Springer-Verlag, New York, 1988, pp. 80–87, and CROCHEMORE, M. String-matching with constraints. In Chytil, Juniga, and Koubeck, eds., *Mathematical Foundations of Computer Science 1988*. Springer-Verlag, New York, 1988, pp. 44–58).

This work was supported by PRC Math-Info.

Authors' address: LITP, Institut Blaise Pascal, Université Paris 7, 2 Place Jussieu 75251, Paris cedex 05, France.

The recognition of patterns in strings is related to the corresponding problem on images. Indeed, some algorithms on words can be generalized to two-dimensional arrays (see [20], for instance). Also extracting the contour of a two-dimensional object leads to a one-dimensional object that can again be considered as a string of symbols. The patterns recognized in this string give in return valuable information on the two-dimensional object. More generally string-processing algorithms can be used on curves, that is to say one-dimensional objects in the plane or in space.

In this paper, we present a new method for string-matching. The general problem of pattern matching in strings can be handled by standard and well-known techniques from automata theory. As a particular case of pattern matching in strings, the problem of string-matching simply consists in locating a given word, the *pattern*, within another one, the *text*. It has been extensively considered and is still an active field of research. This study is both concerned with optimization for practical purposes and also with theoretical considerations. For practical purposes, it is interesting to develop algorithms that are fast, require little memory and limited buffering, and operate in real time. This may be especially meaningful in the case of words coding images or contours since, in this case, the pattern can be very long. From the theoretical point of view, it is interesting to know the lower bounds achievable in such a problem. Those lower bounds correspond to various quantities such as time and space and it is not always possible to optimize them all at the same time. Another theoretical incentive comes from the theory of programming. Indeed, string-matching algorithms have led to computer programs that are now considered as paradigms in the theory of program development. Finally, extensions of the string-matching problem have been studied. In particular, several approximate string-matching algorithms have been proposed [23, 32].

The classical algorithms can be divided into two families. Roughly speaking, in the first family, the pattern x is considered as fixed and the text t as variable. The converse point of view is adopted in the second family. The first family of algorithms contains the well-known algorithms of Knuth, Morris, and Pratt (KMP) on the one hand and of Boyer and Moore (BM) on the other hand (see [8] and [21]). These algorithms were studied and improved by several authors (see [2], [3], [15], [16], [18], [27], [30], and [33]). The second family is based upon the notion of a suffix tree due to Weiner [34]. An efficient algorithm to compute suffix trees was devised by McCreight [24]. Later on, this construction was superseded by the factor transducer construction (see [6] and [10]).

The new algorithm presented here belongs to the first family. From the practical viewpoint, its merits consist of requiring only constant additional memory space. It can therefore be compared with the algorithm of [17], but it presents the advantage of being faster and simpler. From the theoretical viewpoint, its main feature is that it makes use of a deep theorem on words known as the Critical Factorization Theorem due to Cesari, Duval, Vincent, and Lothaire (see [9], [13], and [22]). It is also amusing that the new algorithm, which operates in a two-way fashion, can be considered as a compromise between KMP and BM.

The paper is divided into six sections: The first section provides an introduction. In the second section, we introduce a basic version of our two-way algorithm. The algorithm is linear in time and uses constant space. It requires a

precomputation of a critical factorization and of the period of the pattern. In the third section, we give a new proof of the critical factorization theorem that is adapted to our purpose. The computation of a critical factorization relies on the knowledge of the maximal suffix of a word. The computation of such maximal suffixes is dealt with in the fourth section. In the fifth section, we show how to modify the two-way algorithm to avoid the use of the exact value of the period. The result is a new linear algorithm using constant space. The sixth section contains our conclusions.

2. *String-Matching*

We shall discuss string-matching algorithms. The strings processed by these algorithms are called *words*. These words will be usually denoted as arrays $x = x[1]x[2] \cdots x[n]$. The integer n is called the length of the word x and denoted by $|x|$. We denote by a mere juxtaposition the concatenation of two words.

Among all string-matching algorithms developed up to now, two of them are particularly famous and efficient. They are known as Knuth, Morris, and Pratt's algorithm and Boyer and Moore's algorithm. Let us briefly recall how they work. Let x be the word that is searched for inside an a priori longer word t. The word x is called the *pattern* and t is the *text*. The output of a string-matching algorithm is the set of positions of occurrences of x in t, say

$$P(x, t) = \{k \in \mathbf{N} \mid 0 \leq k \leq |t| - |x| \text{ and } t[k+i] = x[i],\ 1 \leq i \leq |x|\}.$$

Usual string-matching algorithms initially check whether x appears at the left end of t and repeat this process at increasing positions. The word x can thus be visualized as shifted to the right until it reaches the right end of t. Shifts must be as long as possible in order to save time.

In Knuth, Morris, and Pratt's algorithm, the letters of x are checked against the letters of t in a left to right scan of x until its right end is reached, if x occurs at that position, or until a mismatch is met (see Figure 1). If u is the longest prefix of x recognized at the current position, then the shift is made according to both a period of u and the letter of t that causes the mismatch. Hence, a shift function, whose domain is the set of prefixes of x, is precomputed.

In Boyer and Moore's algorithm, the letters of x are scanned from right to left, and x is shifted according to both the periods of its suffixes and the letter of t that causes a possible mismatch (see Figure 2). Proceeding in that way increases the length of shifts in the average. For instance, a letter of t that does not occur in x leads to a shift $|x|$ positions to the right, the best possible without missing an occurrence of x.

During the search for x in Boyer and Moore's algorithm, either the result of some comparisons is forgotten, at the cost of increasing the time of the algorithm, or all comparisons are memorized, but this complicates the preprocessing. Tricky versions of this algorithm have good maximal time complexity, using a number of comparisons bounded by twice the length of t [3, 15]. The number of possible configurations met during the preprocessing is subject to a conjecture stated in [21], that is still open.

The number of letter comparisons used in Knuth, Morris, and Pratt's algorithm is also bounded by $2|t|$. But the two algorithms greatly differ on the

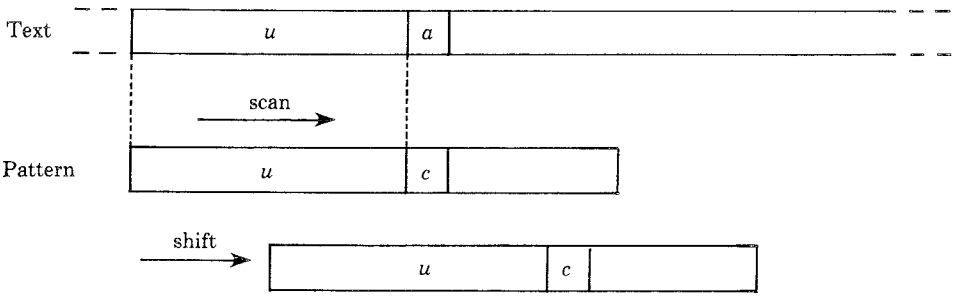

FIG. 1. Knuth, Morris, and Pratt approach.

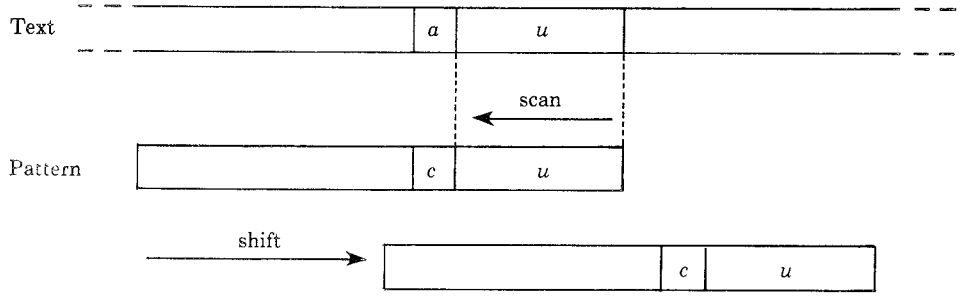

FIG. 2. Boyer and Moore approach.

minimum number of letter comparisons needed to compute $P(x, t)$. It is in fact $|t|$ for Knuth, Morris, and Pratt's algorithm but it becomes $|t|/|x|$ for Boyer and Moore's algorithm. Both algorithms use linear additional memory space for their shift functions on the word x.

We describe an algorithm that, in some sense, may be considered as intermediate between Knuth, Morris, and Pratt's algorithm and Boyer and Moore's algorithm. Our algorithm has the following properties:

(i) It is linear in time $O(|t| + |x|)$, as KMP and BM, with a maximum number of letter comparisons bounded by $2|t| + 5|x|$ compared to $2|t| + 2|x|$ for KMP and $2|t| + f(|x|)$ for BM, where f depends on the version of this algorithm.

(ii) The minimal number of letter comparisons used during the search phase (excluding the preprocessing of the pattern) is $2|t|/|x|$ compared to $|t|$ for KMP and $|t|/|x|$ for BM.

(iii) The memory space used, additionally to the locations of the text and the pattern, is constant instead of $O(|x|)$ for both KMP and BM.

Another algorithm with similar features has been designed by Galil and Seiferas (GS) [17]. Ours is conceptually more simple and consequently easier to analyze. The number of comparisons is also smaller although the precise analysis of the number of comparisons used by GS is difficult.

The new algorithm uses some notions on the structure of words that we now introduce. Let x be a word on an alphabet A. A strictly positive integer p is

called a *period* of x if

$$x[i] = x[i + p],$$

whenever both sides are defined. In other terms, p is a period of x if two letters of x at distance p always coincide. The smallest period of x is called *the period* of x and is denoted by $p(x)$. One has the inequalities

$$0 < p(x) \leq |x|.$$

Let x be a word and l be an integer such that $0 \leq l \leq |x|$. An integer $r \geq 1$ is called a *local period* of x at position l if one has

$$x[i] = x[i + r]$$

for all indices i such that $l - r + 1 \leq i \leq l$ and such that both sides of the equality are defined (see Figure 3). The *local period* of x at position l is the smallest local period at position l. It is denoted $r(x, l)$. It is an easy consequence of the definitions that

$$1 \leq r(x, l) \leq p(x).$$

It is convenient to reformulate the definition of a local period as follows: An integer r is a local period of x at position l if and only if there exists a word w of length r such that one of the four conditions is satisfied:

(i) $x = x'wwx''$ with $|x'w| = l$,
(ii) $x = x'wu$ with $|x'w| = l$ and u prefix of w,
(iii) $x = vwx''$ with $|v| = l$ and v suffix of w,
(iv) $x = vu$ with $|v| = l$, v suffix of w and u prefix of w.

The case (i) is represented on Figure 3. The other cases correspond to a position close to the ends of x or equivalently to an overflow of w. The relation between both definitions is given by

$$w[i] = x[l + i] \quad \text{or} \quad w[i] = x[l - r + i]$$

(for $1 \leq i \leq r$), according to which expression is defined on the right-hand side.

Our algorithm relies on the following result due to Cesari, Vincent, and Duval (see [22] for precise references).

THEOREM (CRITICAL FACTORIZATION THEOREM). *For each word x, there exists at least one position l with $0 \leq l < p(x)$ such that*

$$r(x, l) = p(x).$$

A position l such that $r(x, l) = p(x)$ is called a *critical position* of x. For instance, the word

$$x = abaabaa$$

has period 3. It has three critical positions, namely 2, 4, 5. We shall prove the Critical Factorization Theorem in Section 3 and give an algorithm allowing to compute a critical position in Section 4. Before coming to this description, we present our application of this theorem to string-matching. We begin with an informal description.

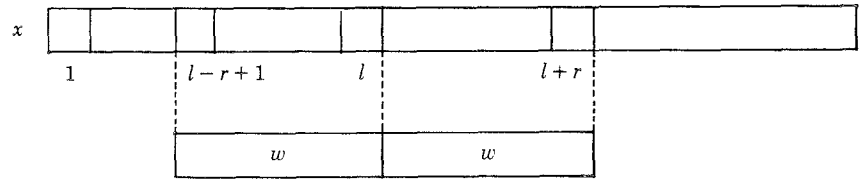

FIG. 3. A local period r at position l.

Our string-matching algorithm matches the letters of the pattern x and of the text t in both directions. The starting point of the scan is given by a critical position l of x that satisfies

$$l < p(x),$$

where $p(x)$ is the period of x. We denote by x_l the prefix of length l of x and we define x_r as the suffix of x such that $x = x_l x_r$.

To discover an occurrence of the pattern x at a given position of the text t, the algorithm conceptually divides the search into two successive phases. The first phase consists in matching x_r only against t. The letters of x_r are scanned from left to right.

When a mismatch is found during the first phase, there is no second phase and the pattern is shifted to the right. The shift brings the critical position to the right of the letter of the text that caused the mismatch.

If no mismatch happened during the first phase, that is, when an occurrence of x_r is found in t, the second phase starts. The left part x_l of the pattern is matched against the text. The word x_l is scanned from right to left as in the Boyer and Moore approach. If a mismatch occurs during the scanning, the pattern is shifted a number of places equal to the period of x (see Figure 4). After such a shift, some prefix of the pattern coincides with the text. This prefix is memorized in order to possibly increase the length of the next shift.

Example. Let $x = a^n b$ and let $t = aaa \cdots$ be an arbitrarily long repetition of a's. The algorithm of Knuth, Morris, and Pratt shifts the pattern according to the Figure 5(a) and so do Boyer and Moore's and our algorithm. The number of letter comparisons is $2|t| - n$ for the first algorithm and $|t| - n$ for the two others. Indeed, since the unique critical position of $a^n b$ is n, both algorithms attempt to match the last letter of x against the letters of t. If we replace $t = aaa \cdots$ by $t = bbb \cdots$, then the sequence of shifts is shown on Figure 5(b) for KMPs and on Figure 5(c) for BM algorithms. Our algorithm gives rise to the same sequence of shifts and letter comparisons as BMs. This time, each mismatch is detected in the left part of the factorization (a^n, b) and the pattern is shifted according to the period of x; that is, $n + 1$ places to the right. □

We shall now describe more formally our algorithm. We temporarily suppose that the period $p(x)$ of the pattern x and a critical position $l \leq p(x)$ have been computed previously. We shall discuss in the next sections how to compute them both.

The algorithm is presented as a function computing the set $P(x, t)$ of positions of x in t. It uses four local variables i, j, s, and pos. The variables i

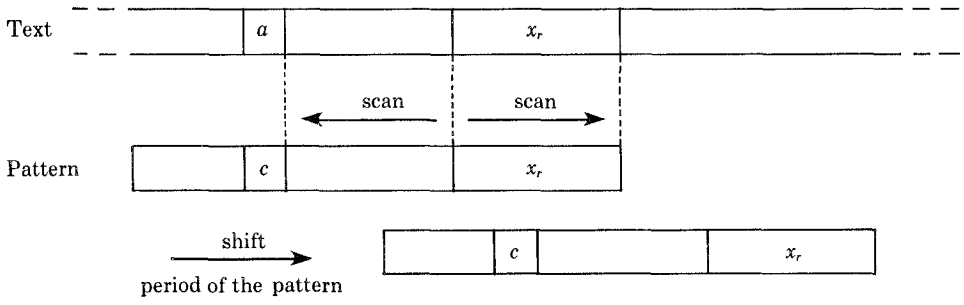

FIG. 4. String-matching with critical position.

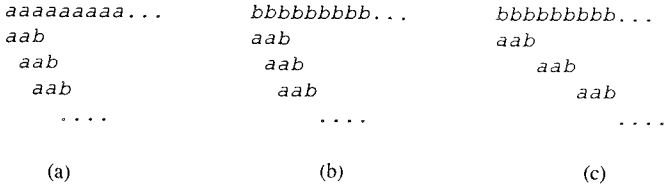

FIG. 5. Matching $x = aab$ in $t = aaa \cdots$ or $t = bbb \cdots$. (a) KMP & BM. (b) KMP. (c) BM.

and j are used as cursors on the pattern to perform the matching on each side of the critical position respectively (see Figure 6). The variable s is used to memorize a prefix of the pattern that matches the text at the current position, given by the variable *pos* (see Figure 7). Variable s is updated at every mismatch. It can be set to a nonzero value only in the case of a mismatch occurring during the scan of the left part of the pattern.

The scheme of the string-matching algorithm is shown on Figure 8.

Before proving the correctness of the algorithm, we first give a property of a critical position of a word.

LEMMA 2.1. *Let l be a critical position of a nonempty word x. Let r be a local period at the position l, such that $r \leq max(l, |x| - l)$. Then r is a multiple of $p(x)$.*

PROOF. (See Figure 9). Assume that $r \leq |x| - l$. Let r' be the remainder of r divided by $p(x)$. Since $l + r \leq |x|$, the word $x[l+1] \cdots x[l+r]$ has period $p(x)$. Then, we have the equality $x[l+1] \cdots x[l+r'] = x[l+r-r'+1] \cdots x[l+r]$, which means that if $r' > 0$, it is a local period at l. But $r' < p(x)$ gives a contradiction with the fact that l is a critical position. The only possibility is thus $r' = 0$, which is equivalent to say that r is a multiple of $p(x)$. The case $r \leq l$ is symmetrical. □

PROPOSITION 2.2. *On input words x and t, the function* "POSITIONS" *computes the set $P(x, t)$ of positions of the pattern x inside the text t.*

PROOF. The execution of the function stops because of the three following facts:

—At each step of the "while" loop (line 1), one of the statements at lines 4 or 8 is executed.

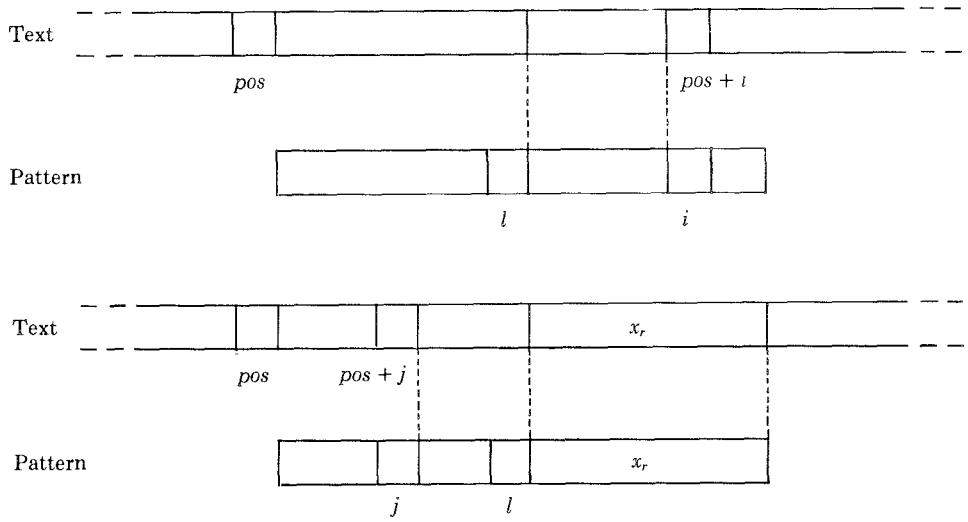

FIG. 6. The role of variables *pos*, *i*, *j*.

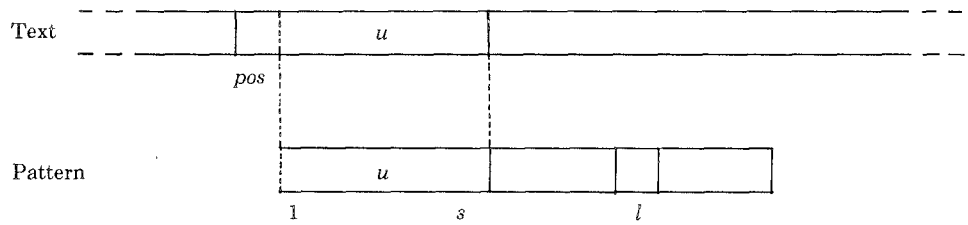

FIG. 7. The role of variable *s*.

—The variable *pos* is strictly incremented at line 4 or 8 because the value of $i - l$ is strictly positive and $p > 0$.
—Then the successive values of *pos* are in strictly increasing order and bounded by $|t|$.

This proves the termination of the algorithm. We now show its correctness. Let $q_1, q_2, \ldots, q_K$ be the successive values of the variable *pos* during a run of the function "POSITIONS" on inputs x and t. Let Q be the final value of the variable P. We prove the conclusion, say $P(x, t) = Q$, by showing that

(i) the algorithm detects any q_k that is a position of an occurrence of x in t (i.e., $P(x, t) \cap \{q_1, q_2, \ldots, q_K\} = Q$), and that
(ii) any position of an occurrence of x in t is some q_k (i.e., $P(x, t) \subseteq \{q_1, q_2, \ldots, q_K\}$).

We first prove that the following property is an invariant of the main "while" loop (line 1):

$$x[s'] = t[pos + s'], \qquad 1 \leq s' \leq s.$$

This is initially true since s is set to null. For the same reason, this is true when a step of the main loop ends with the statement "$s \leftarrow 0$".

```
      function POSITIONS (x, t):
      (p is the period of x and l is a critical position such that l < p)
        P := ∅; pos ← 0; s ← 0;
1       while (pos + |x| ≤ |t|) do {
2         i ← max(l, s) + 1;
3         while (i ≤ |x| and x[i] = t[pos + i]) do i ← i + 1;
          if (i ≤ |x|) then {
4           pos ← pos + max(i − l, s − p + 1);
            s ← 0;
          } else {
5           j ← l;
6           while (j > s and x[j] = t[pos + j]) do j ← j − 1;
7           if (j ≤ s) then add pos to P;
8           pos ← pos + p;
9           s ← |x| − p;
          } end if
        } end while
        return (P);
      end function.
```

FIG. 8. Constant-space string-matching algorithm.

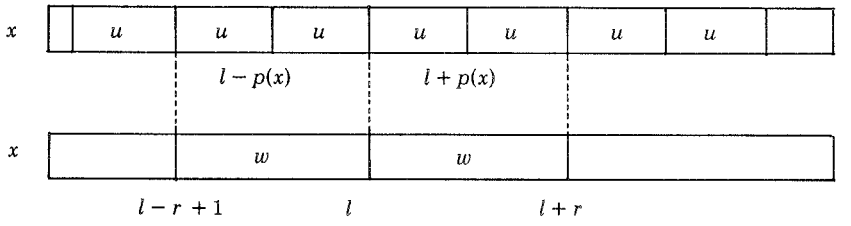

FIG. 9. A local period r at a critical position l.

Consider a step of the loop that ends with the statement at line 9. Let q_{k+1} be the value of *pos* at the end of the step. In this situation, the loop at line 3 must have stopped with condition "$i \leq |x|$" false, which means that

$$x[i'] = t[q_k + i'], \qquad l < i' \leq |x|.$$

Since $q_{k+1} = q_k + p$ and $l < p$ we get

$$x[i'] = t[q_{k+1} + i'], \qquad 0 < i' \leq |x| - p.$$

The property is then true at the end of the step, because its last statement at line 9 gives the value $|x| - p$ to s.

PROOF OF (i). Let q in $P(x, t) \cap \{q_1, q_2, \ldots, q_K\}$. While *pos* has value q, since $q \in P(x, t)$, all letter comparisons at lines 3 and 6 succeed and condition "$j \leq s$" is true at line 7. So q is added to P and then $q \in Q$.

Let q in Q. Since q is a value of *pos* put in P at line 7, during that step all letter comparisons at lines 3 and 6 have succeeded:

$$x[i'] = t[q + i'], \qquad s < i' \leq |x|.$$

By the invariant property of s, this implies $q \in P(x, t)$.

PROOF OF (ii). We prove that no q strictly between two consecutive values of *pos* can be in $P(x, t)$. Let q be a position such that $q_k < q$ and $q \in P(x, t)$. Consider the step of the main "while" loop where the initial value of *pos* is q_k. Let i' be the value assigned to the variable i by the statement at line 3, and let s' be the value of s at the beginning of the step. We now consider two cases according to a mismatch occurring in the right part or in the left part of the pattern. In both cases, we use the following argument: If, after a mismatch, the shift is too small, then it is a multiple of the period of the pattern, which implies that the same mismatch recurs.

Case 1 (see Figure 10). If $i' \leq |x|$, a mismatch has occurred at line 3, and we have:

$$x[l+1] \cdots x[i'-1] = t[q_k + l + 1] \cdots t[q_k + i' - 1],$$
$$x[i'] \neq t[q_k + i'].$$

Let w be the word $t[q_k + l + 1] \cdots t[q + l]$.

If $q < q_k + i' - l$, the above equality implies

$$x[l+1] \cdots x[l + q - q_k] = w.$$

Since $q \in P(x, t)$, the suffixes of length $\max(1, l + 1 - q + q_k)$ of w and $x[l + 1 - q + q_k] \cdots x[l]$ coincide. The quantity $q - q_k$ is then a local period at the critical position l and, by the Lemma 2.1, $q - q_k$ is a multiple of the period of x. So, $x[i'] = x[i' - q + q_k]$. But $q \in P(x, t)$ also implies $x[i' - q + q_k] = t[q + i' - q + q_k]$, which gives a contradiction with the mismatch $x[i'] \neq t[q_k + i']$. This proves $q \geq q_k + i' - |x_l|$.

Assume $q < q_k + s' - p(x) + 1$. Let w be $x[1] \cdots x[p(x)]$. Since $q \in P(x, t)$, $w = t[q + 1] \cdots t[q + p(x)]$, and then w is a suffix of $t[q_k + 1] \cdots t[q_k + q - q_k + p(x)]$. This latter word is a prefix of x because $q - q_k + p(x) < s' + 1$ and because of the invariant property satisfied by s. Thus, $q - q_k$ is a multiple of the period of x and the mismatch between $x[i']$ and $t[q_k + i']$ remains between $x[i' - q + q_k]$ and $t[q_k + i']$. This is a contradiction and proves $q \geq q_k + s' - p(x) + 1$.

So far, we have proved that, if a mismatch occurs at line 3, q is not smaller than $\max(q_k + i - l, q_k + s' - p(x) + 1)$. This last quantity is q_{k+1} (see line 4), and then $q \geq q_{k+1}$.

Case 2 (see Figure 11). If no mismatch is met at line 3, the right part of x occurs at position $q_k + l$ of the text t. The word $w = t[q_k + l + 1] \cdots t[q + l]$ then occurs at the right of position l. Since $q \in P(x, t)$, it also occurs at the left of position l. Thus, $|w|$ is a local period at the critical position l, and then $|w| \geq p(x)$. We get $q - q_k \geq p(x)$. Since the statement at line 8 yields $q_{k+1} = q_k + p(x)$, in this second case, again, the inequality $q \geq q_{k+1}$ holds.

This completes the proof of assertion (ii) and ends the proof of the proposition. □

The time complexity of the function "POSITIONS" is proportional to the number of comparisons between the letters x and t (lines 3 and 6). This number is bounded by $2|t|$ as shown by the following proposition:

PROPOSITION 2.3. *The computation of the set of positions of a word x in a text t of length m by the function* "POSITIONS" *uses less than* $2m$ *letter comparisons.*

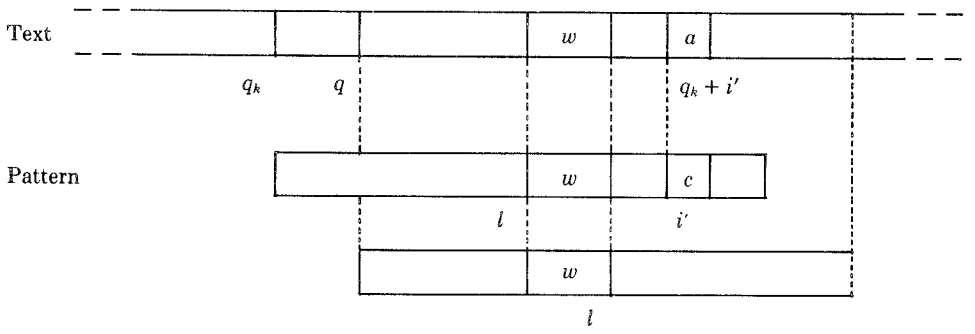

FIG. 10. A shift after a mismatch occurring in the right part of the pattern.

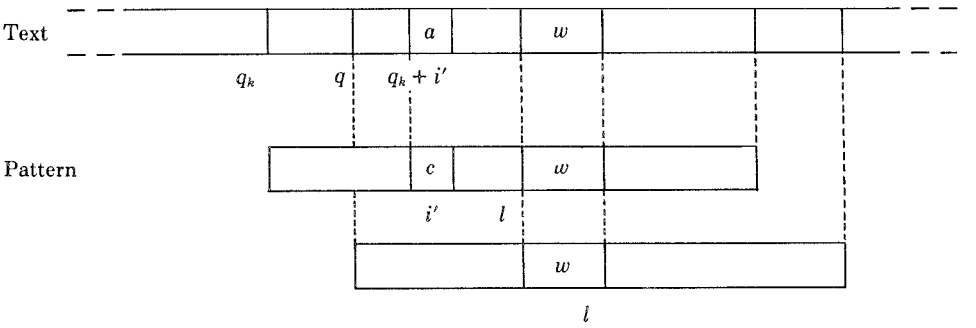

FIG. 11. A shift after a mismatch occurring in the left part of the pattern.

PROOF. We first prove the following assertion: Each letter comparison done at line 3 strictly increases the value of $pos + i$. This is obvious if the letters $x[i]$ and $t[pos + i]$ coincide and "$i < |x|$" holds.

If the letters coincide but $x[i]$ is the last letter of x (no mismatch in the right part of the pattern), the variable i is increased by one unit at line 3. After that, the variable pos is increased by p at line 8, and the variable i is decreased by at most p (thanks to variable s), during the next step of the main loop, at line 2. Then, the assertion holds.

If the letters do not coincide (then a mismatch occurs in the right part of the pattern), let i' be the value of the variable i when the execution of the statement at line 3 stops. The variable pos is then increased by at least $i' - l$, at line 4, and the variable i is decreased by at most $i' - l + 1$, during the next step of the main loop, at line 2. Thus, the assertion holds again.

Hence, the number of letter comparisons done at line 3 is at most $|t| - |x_l|$, because expression $pos + i$ has initial value $|x_l| + 1$ and terminal value $|t|$ in the worst case.

At line 6, comparisons are done on letters of the left part of the pattern. The next instruction may be considered as a shift of x, $p(x)$ places to the right. Since, by assumption, the length of x_l is less than $p(x)$, two comparisons performed at line 6 are done on letters of t occurring at different positions inside t. Then, at most, $|t|$ letter comparisons are globally done at line 6.

This gives the upper bound $2|t|$ to the number of letter comparisons. □

COROLLARY 2.4. *According to the RAM model, the maximum time complexity of function "POSITIONS" is $O(m)$ on an input text of length m. The function requires seven extra integer memory locations in addition to the arrays x and t.*

PROOF. The time complexity is proportional to the number of letter comparisons. The results are then a consequence of Proposition 2.3.

Three memory locations only are needed for the variables pos, s, i, j, because i and j can be associated with the same storage location. The quantities $p, l, |x|, |t|$ need four more memory locations. □

3. *The Critical Factorization Theorem*

We now present a proof of the Critical Factorization Theorem. This proof gives a method both practically and algorithmically simple to compute a critical position. The method relies on the computation of maximal suffixes, which will be presented in the next section.

For the convenience of the exposition, we make our notation concerning words more precise. Let A be a finite alphabet, and let A^* be the set of words on the alphabet A. We shall denote by ϵ the empty word. We denote by $|w|$ the length of a word w. Thus, $|\epsilon| = 0$. We write $w[i]$ the ith letter of the word w.

Let x be a word on A. We say that a pair (u, v) of words on A is a *factorization* of x if $x = uv$. The factorization (u, v) of x then defines a *cutpoint* inside the word x. The word u is called a *prefix* of x and v is called a *suffix*. A prefix of v is called a *factor* of x, occurring in x at position $|u|$. A word u that is both a prefix and a suffix of x is called a *border* of x. An *unbordered* word is a nonempty word x that admits no border except itself and the empty word. One may verify that $p(x)$ is the difference of $|x|$ and the length of the longest proper border of x.

Given a factorization (u, v) of x, a *local period* at (u, v) is the same as a local period at position $|u|$ in the terminology of Section 2. The minimum local period at (u, v) is denoted by $r(u, v)$ and the Critical Factorization Theorem can be restated as follows.

THEOREM (CRITICAL FACTORIZATION THEOREM). *For each word x, there exists at least one factorization (u, v) of x such that*

$$r(u, v) = p(x).$$

Moreover u can be chosen with $|u| < p(x)$.

A factorization (u, v) such that $r(u, v) = p(x)$ is called a *critical factorization* of x. For instance, the word

$$x = abaabaa$$

has period 3. It has three critical factorizations, namely,

$$(ab, aabaa), \qquad (abaa, baa), \qquad (abaab, aa).$$

There exist several available proofs of this result. All of them lead to a more precise result asserting the existence of a critical factorization with a cutpoint in each factor of length equal to the period of x.

A weak version of the theorem occurs if one makes the additional assumption that the inequality

$$3p(x) \leq |x|$$

holds. Indeed, in this case, one may write $x = lwwr$, where $|w| = p(x)$ and w is chosen minimal among its cyclic shifts. This means, by definition, that w is a Lyndon word (see [22]). One can prove that a Lyndon word is unbordered. Consequently, the factorization

$$(lw, wr)$$

is critical. This version of the theorem is the argument used in Galil and Seiferas's algorithm (see Lemma 3 in [17]) to build a string-matching algorithm using restricted memory space. This result is used in [17] to prove a decomposition theorem according to which, any word x has a factorization (u, v) such that v does not have two prefixes that are fourth powers in a nontrivial way.

In the sequel, we shall be interested in the computation of a critical factorization of a given word. Among the existing proofs, one relies on the property that if $x = ayb$ with a, b being two letters then a critical factorization of x either comes from a critical factorization of ay or from a critical factorization of yb [22]. This leads to a quadratic algorithm. Another proof given in [13] relies on the notion of a Lyndon factorization of a word. It leads, via the use of a linear string-matching algorithm to a linear algorithm for computing a critical factorization.

We present here a new proof of the critical factorization theorem. This proof leads to a relatively simple linear algorithm that in addition, uses only constant additional memory space.

Each ordering $\leq$ on the alphabet A extends to an *alphabetical ordering* on the set A^*. It is defined as usual by $x \leq y$ if either x is a prefix of y or if

$$x = lar, \qquad y = lbs$$

with l, r, s words of A^* and a, b two letters such that $a < b$.

THEOREM 3.1. *Let $\leq$ be an alphabetical ordering and let $\subseteq$ be the alphabetical ordering obtained by reversing the order $\leq$ on A.*

Let x be a nonempty word on A. Let v (resp., v') be the alphabetically maximal suffix of x according to the ordering $\leq$ (resp., $\subseteq$). Let $x = uv = u'v'$.

If $|v| \leq |v'|$, then (u, v) is a critical factorization of x. Otherwise, (u', v') is a critical factorization of x. Moreover, $|u| < p(x)$ and $|u'| < p(x)$.

The proof of Theorem 3.1 relies on the following lemma:

LEMMA 3.2. *Let v be the alphabetically maximal suffix of x and let $x = uv$. Then no nonempty word is both a suffix of u and a prefix of v.*

PROOF. Let w be a word which is both a suffix of u and a prefix of v. Let $v = wt$. By the definition of v, we have $wv \leq v$ and $t \leq v$. The first inequality can be written $wwt \leq wt$ and this implies $wt \leq t$. The second inequality can be written $t \leq wt$. We then obtain $t = wt$, where $w = \epsilon$. □

PROOF OF THE THEOREM. First, observe for later use that the intersection of the orderings $\leq$ and $\subseteq$ is the prefix ordering. Equivalently, for any words w and w',

$$w \leq w' \quad \text{and} \quad w \subseteq w'$$

if and only if w is a prefix of w'.

We first dispose of the case where the word x has period 1, that is to say, when x is a power of a single letter. In this case, any factorization of x is critical.

We now suppose that $|v| \leq |v'|$ and prove that (u, v) is a critical factorization. The other case is symmetrical. Let us prove first that $u \neq \epsilon$. Indeed, let $x = ay$ with a in A. If $u = \epsilon$, then $x = v = v'$ and both inequalities $y \leq x$ and $y \subseteq x$ are satisfied by the definition of v and v'. Thus, y is a prefix of x, where $p(x) = 1$ contrary to the hypothesis.

Let r be the local period at (u, v). By Lemma 3.2, we cannot simultaneously have $r \leq |u|$ and $r \leq |v|$. Moreover, since v is alphabetically maximal, it cannot be a factor of u. Hence, $r > |u|$ since $r \leq |u|$ would imply $r > |v|$ and v factor of u. Let z be the shortest word such that v is a prefix of zu or vice versa zu is a prefix of v. Then, $r = |zu|$. We now distinguish two cases according to $r > |v|$ or $r \leq |v|$.

Case $r > |v|$ (see Figure 12). In this situation, by the definition of r, the word u cannot be a factor of v. The integer $|uz|$ is a period of uv since uv is a prefix of $uzuz$. The period of uv cannot be shorter than $|uz|$ because this quantity is the local period at (u, v). Hence, $p(uv) = |uz| = r$. This proves that the factorization (u, v) is critical.

Case $r \leq |v|$ (see Figure 13). The word u is a factor of v. Since $|zu|$ is the local period at (u, v), as in the previous case, we only need to prove that $|zu|$ is a period of x.

Let $u = u'u''$ and $v = zuz'$. By the definition of v', the suffix $u''z'$ of uv satisfies

$$u''z' \subseteq v' = u''v,$$

hence, $z' \subseteq v$. By the definition of v, we also have $z' \leq v$. By the observation made at the beginning of the proof, these two inequalities imply that z' is a prefix of $v = zuz'$. Hence, z' is a prefix of a long enough repetition of zu's. Since $x = uzuz'$, this shows that $|uz|$ is a period of x.

Since, as proved above, $|u|$ is less than the local period r, we get $|u| < p(x)$ because the factorization is critical. We also get $|u'| < p(x)$ when $|u'| < |u|$. The same argument holds symmetrically when this latter expression fails. □

According to Theorem 3.1, the computation of a critical factorization reduces to that of maximal suffixes. More accurately, it requires the computation of two maximal suffixes corresponding to reversed orderings of the alphabet.

4. *Maximal Suffixes*

In this section, we describe an efficient algorithm to compute the alphabetically maximal suffix of a given word. It is a consequence of several known algorithms that this computation can be done in linear time. One may use the

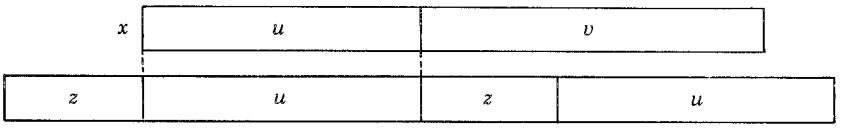

FIG. 12. Case $r > |v|$.

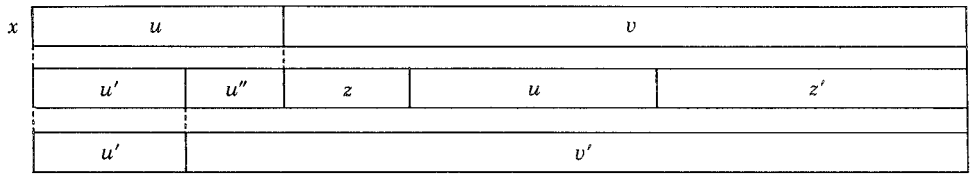

FIG. 13. Case $r \leq |v|$.

suffix tree construction [24], the factor automaton construction [6, 10], or also the Lyndon factorization [14]. One may also adapt the least circular word algorithm of [7] and [29]. We describe here an algorithm which is essentially the same as the one described in [14] but slightly more simple.

We consider a fixed ordering on the alphabet A. We denote by $max(x)$ the suffix of x which is maximal for alphabetic ordering. Let u, v be words and $e \geq 1$ be an integer such that $max(x) = u^e v$ with $|u| = p(max(x))$ and where v is a proper prefix of u (recall from Section 2 that $p(y)$ denotes the period of the word y). We denote

$$per(x) = u, \qquad rest(x) = v.$$

Note that $rest(x)$ is a border of $max(x)$.

We now prove a series of statements that, all together, give a recursive scheme allowing the computation of the maximal suffix of a given word. We shall also describe afterwards a nonrecursive version of the algorithm.

First of all, for any word x and any letter a, the maximal suffix of xa is a suffix of $max(x)a$ since, for any suffix w of x longer than $max(x)$, we have $w < max(x)$ with w not a prefix of $max(x)$, where $wa < max(x)a$.

PROPOSITION 4.1. *Let x be a word and a be a letter. Let a' be the letter such that $rest(x)a'$ is a prefix of x. We have*

$$max(xa) = \begin{cases} max(x)a & \text{if } a \leq a', \\ max(rest(x)a) & \text{if } a > a', \end{cases}$$

$$per(xa) = \begin{cases} max(x)a & \text{if } a < a', \\ per(x) & \text{if } a = a', \\ per(rest(x)a) & \text{if } a > a', \end{cases}$$

$$rest(xa) = \begin{cases} \epsilon & \text{if } a < a' \text{ or } (a = a' \text{ and } rest(x)a = per(x)), \\ rest(x)a & \text{if } (a = a' \text{ and } rest(x)a < per(x)), \\ rest(rest(x)a) & \text{if } a > a'. \end{cases}$$

PROOF. We first show that for any proper border w of $max(x)$ longer than $rest(x)wa'$ is a prefix of $max(x)$. Let $u = per(x)$, $v = rest(x)$ and let $e \geq 1$ be such that $max(x) = u^e v$. Let b be the letter such that wb is a prefix of $max(x)$.

First suppose that w is the longest border of $max(x)$, that is to say $max(x) = uw$. We have $per(x) = va'v'$, $w = u^{e-1}v$ and $max(x) = u^{e-1}va'v'v = wa'v'v$. Hence, $b = a'$ in this case (see Figure 14).

If w is not the longest proper border of $max(x)$, then w is a proper suffix of $u^{e-1}v$ (see Figure 15). By the above argument, $u^{e-1}va'$ is a prefix of $max(x)$. Hence, wa' is a factor of $max(x)$, where $wb \geq wa'$, or equivalently $b \geq a'$. We now show that the converse inequality $b \leq a'$ also holds. Since v is a suffix of w and since wb is a prefix of $max(x)$, vb is a factor of $max(x)$. Hence, $va' \geq vb$ or equivalently $a' \geq b$. The conclusion is $a' = b$ as promised.

We now come to the proof of the three statements of Proposition 4.1. We treat three cases separately.

(i) Let us first suppose that $a < a'$. We show that in this case $max(x)a$ is an unbordered word. Let w be a border of $max(x)$. If w is shorter than $rest(x)$, then it is a border of $rest(x)$ (see Figure 16). Let $rest(x) = wbw'$ with b a letter. Since wb is a prefix of $max(x)$ and wa' is a factor of $max(x)$ we have $wb \geq wa'$, where $b \geq a'$ and therefore $b > a$. This proves that wa cannot be a prefix of $max(x)$.

If w is a border of $max(x)$ longer than $rest(x)$, then, since $a \neq a'$, wa cannot be a prefix of $max(x)$, and this concludes the proof that $max(x)a$ is unbordered.

We are now able to prove that $max(x)a$ is maximal among its suffixes. Let w be a suffix of $max(x)$. Since, if $max(x) > w$ without w being a prefix of $max(x)$, we also have $max(x)a > wa$, where the conclusion follows. This shows that

$$max(xa) = max(x)a, \qquad per(xa) = max(x)a, \qquad rest(xa) = \epsilon.$$

(ii) Let us now suppose that $a = a'$. It is then a direct consequence of the definitions that

$$max(xa) = max(x)a; \qquad per(xa) = per(x)$$

and that

$$rest(xa) = \begin{cases} \epsilon & \text{if} \quad rest(x)a = per(x) \\ rest(x)a & \text{otherwise.} \end{cases}$$

(iii) Let us finally suppose that $a > a'$. Let $wa = max(xa)$. Since w is a suffix of $max(x)$, we have $max(x) > w$. This forces w to be a prefix of $max(x)$, and hence a border of $max(x)$, since otherwise, $max(x) > w$ would imply $max(x)a > wa$, a contradiction.

The border w cannot be strictly longer than $rest(x)$ since, as shown above, wa is not a prefix of $max(x)$ when $a \neq a'$.

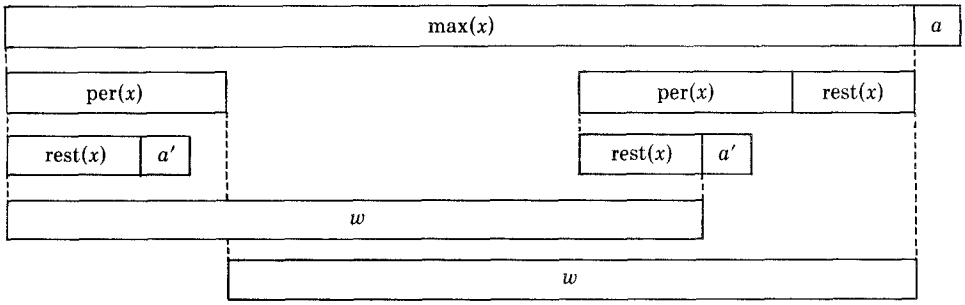

FIG. 14. The longest proper border of $max(x)$.

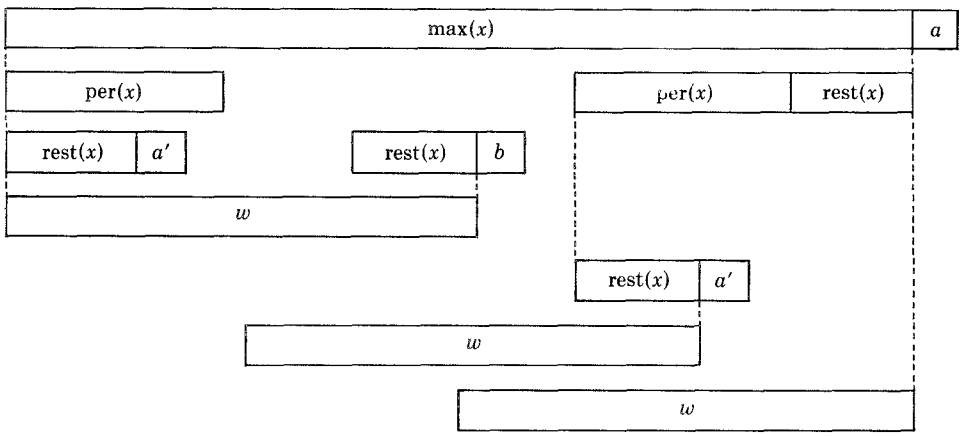

FIG. 15. A border of $max(x)$ longer than $rest(x)$ but not maximal.

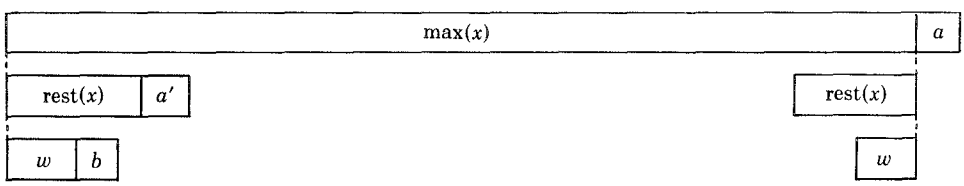

FIG. 16. A border of $max(x)$ shorter than $rest(x)$.

Therefore, w is a suffix of $rest(x)$ and this proves the formulas:

$$max(xa) = max(rest(x)a),$$
$$per(xa) = per(rest(x)a),$$
$$rest(xa) = rest(rest(x)a).$$ □

We now give a nonrecursive algorithm that allows the computation of the maximal suffix of a word. It is given in Figure 17 as a function "MAXIMAL-SUFFIX". This function outputs the position of $max(x)$ and its period.

The interpretation of the variables i, j, k occurring in the algorithm of Figure 17 is indicated on Figure 18.

```
function MAXIMAL-SUFFIX (x[1] ··· x[n]):
  i ← 0; j ← 1; k ← 1; p ← 1;
  while (j + k ≤ n) do {
    a' ← x[i + k]; a ← x[j + k];
    if (a < a') then {j ← j + k; k ← 1; p ← j − i;}
    if (a = a') then
      if (k = p) then {j ← j + p; k ← 1;} else k ← k + 1;
    if (a > a') then {i ← j; j ← i + 1; k ← 1; p ← 1;}
  } end while
  return (i, p);
end function.
```

FIG. 17. The alphabetically maximal suffix of a word.

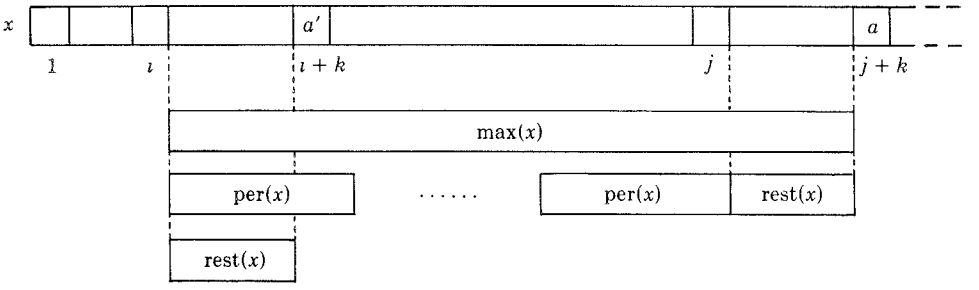

FIG. 18. The role of variables i, j, k.

The integer p is the period of $max(x)$, that is the length of $per(x)$. The integer i is the position of $max(x)$, and j is the position of the last occurrence of $rest(x)$ in $max(x)$.

The correctness of the algorithm is clear by Proposition 4.1. Its time complexity can be analyzed as follows:

PROPOSITION 4.2. *The computation of the function* "MAXIMAL-SUFFIX" *by the algorithm of Figure* 17 *uses less than* $2n$ *letter comparisons on an input word of length* n.

PROOF. We show that after each comparison between letters a and a', the expression

$$i + j + k$$

is increased at least by one unit. Since $i \leq n$ and $j + k \leq n + 1$, we have

$$2 \leq i + j + k \leq 2n + 1,$$

which implies that the number of comparisons performed before the execution of the algorithm stops is at most $2n$. We now look at the three possible cases:

(i) In case $a < a'$, $i + j + k$ is replaced by $i + j + k + 1$.
(ii) In case $a = a'$, $i + j + k$ is also replaced by $i + j + k + 1$.
(iii) In case $a > a'$, $i + j + k$ is replaced by $2j + 2$. But, since we have always $i + k \leq j$, we obtain $i + j + k \leq 2j$. Hence, $i + j + k$ increases at least by 2 units in this case. □

As a consequence of Proposition 4.2, the time complexity of the algorithm of Figure 17 in the worst case is $O(n)$ in the RAM model. The computation requires seven extra memory locations.

It is worth observing that the arguments used in the proof of Proposition 4.1 are essentially the same arguments required to prove the following statement: *Let* x *be a word and let* $x = u^e v$ *with* $|u| = p(x)$ *and* $e \geq 1$; *then* $x = max(x)$ *iff* $u = max(u)$ *and* u *is unbordered.* This statement is closely related to properties of Lyndon words proved in [14]. Indeed, it is equivalent to say that u is a Lyndon word for the reversed alphabetical ordering, and that u is an unbordered word such that $u = max(u)$.

The algorithm developed in this section can be used for the computation of a critical factorization (u, v) of a word x such that $|u| < p(x)$, as we have seen in Section 3. Moreover, the above algorithm computes the period of $max(x)$ and we shall see in Section 5 how this algorithm can be used to approximate the period of the word x.

5. *Computing Periods*

The string-matching algorithm of Section 2 uses the period of the pattern. A previous computation of this period is possible. This precomputation can be made easily by using Knuth, Morris, and Pratt's algorithm on the pattern. This precomputation, combined with both the computation of a critical factorization of the pattern and our string-matching algorithm, leads to an algorithm that is globally linear in time and space. But, since the string-matching algorithm operates in constant space, it is desirable to improve on the precomputation of the period of the pattern, with the aim of obtaining a global algorithm that is in linear time and constant space. There are two ways to achieve that goal. One is a direct computation of the period by an algorithm operating in linear time and constant space. Such an algorithm is described in [12]. It can also be obtained as a consequence of the results in [17]. Our approach here is different. We shall describe a modification of our string-matching algorithm that avoids the use of the period of the pattern, giving an algorithm that is linear time and constant space on the whole. The point is that the exact value of the period of the pattern is actually needed only when it is less than half the length of the pattern. When the period of the pattern is larger, an even simpler string-matching algorithm is provided.

We first show how small periods can be computed in linear time and bounded space. Then, we give a second version of the "function POSITION," called "POSITION-BIS," intended to deal with patterns having no period less than half their length. The complete *two-way string-matching* algorithm, gathering the functions "POSITION," "POSITION-BIS," and "SMALL-PERIOD," is given as a function called "MATCH" in Figure 21.

The algorithm of Figure 19 computes $p(x)$, the period of its input, when it satisfies $p(x) \leq |x|/2$, and otherwise produces a lower bound of $p(x)$ as we shall see in next proposition. The maximum number of letter comparisons used by the algorithm on input x is $4.5|x|$ decomposed as follows: $2|x|$ at line 1, $2|x|$ at line 2 (see Section 4), and $|x|/2$ at line 4. The following statement proves the correctness of the algorithm.

PROPOSITION 5.1. *Let* x *be a nonempty word. Let* (u, v) *be a critical factorization of* x *such that* $|u| < p(x)$. *Let* $v = y^e z$ *with* $e \geq 1$ *and*

```
function SMALL-PERIOD (x):
  n ← LENGTH(x);
1 (l1, p1) ← MAXIMAL-SUFFIX(x, ≤);
2 (l2, p2) ← MAXIMAL-SUFFIX(x, ⊆);
3 if (l1 ≥ l2) then {
    l ← l1; p ← p1;
  } else {
    l ← l2; p ← p2;
  } end if
4 if (l < n/2 and x[1] ··· x[l] is a suffix of x[l + 1] ··· x[l + p]) then {
5   return (p);
  } else {
6   return (max(l, n − l) + 1);
  } end if
end function.
```

FIG. 19. Constant-space computation of small periods

$|y| = p(v)$. *If* $|u| < |x|/2$ *and* u *is a suffix of* y, *then* $p(x) = p(v)$; *otherwise*, $p(x) > max(|u|, |v|)$.

PROOF. If the condition holds true, the word x is a factor of y^{e+2}. Then, $|y| = p(v)$ is a period of x and, since the period of x cannot be less than the period of v, we get $p(x) = p(v)$.

If $|u| \geq |x|/2$, we have obviously $\max(|u|, |v|) = |u|$ and $p(x) > \max(|u|, |v|)$.

Finally consider the case where $|u| < |x|/2$ and u is not a suffix of y. We show that there is no nonempty word w such that wu is a prefix of x. Assume, ab absurdo, that wu is prefix of x. If w is nonempty, its length is a local period at (u, v), and then $|w| \geq p(x) \geq p(v)$. We cannot have $|w| = p(v)$ because u is not a suffix of y. We cannot either have $|w| > p(v)$ because this would lead to a local period at (u, v) strictly less than $p(v)$, a contradiction. This proves the assertion and also shows that the local period at (u, v) is strictly larger than $|v|$. Since $\max(|u|, |v|) = |v|$, we get the conclusion: $p(x) > \max(|u|, |v|)$. □

Actually, we observe that the condition stated in the proposition holds true when the period of x is small, that is, when $p(x) \leq |x|/2$, implying $p(x) = p(v)$, as previously mentioned.

We now present in Figure 20 a simple version of our string-matching algorithm to be used when the period of the pattern is large. This new version differs in the way a mismatch occurring in the second phase is treated. In this situation, instead of shifting the pattern $p(x)$ places to the right, it is only shifted q places to the right with $q \leq p(x)$. The correctness of this modification is obvious and the time complexity is still linear, provided q satisfies some requirement as explained below.

PROPOSITION 5.2. *Let* x *and* t *be words and let* q *be an integer such that* $0 < q \leq p(x)$. *Then, the function* "POSITIONS-BIS", *which uses both the integer* q *and a critical position* l *such that* $l < p(x)$, *computes the set* $P(x, t)$ *of positions of the pattern* x *inside the text* t.

Furthermore, if $q > max(l, |x| - l)$ *the number of letter comparisons used by the algorithm is bounded by* $2|t|$.

```
        function POSITIONS-BIS (x, t):
        (q is an integer such that 0 < q ≤ p(x) and
         l is a critical position such that l < p(x))
          P := ∅; pos ← 0;
1         while (pos + |x| ≤ |t|) do {
2           i ← l + 1:
3           while (i ≤ |x| and x[i] = t[pos + i]) do i ← i + 1;
            if (i ≤ |x|) then {
4             pos ← pos + i − l;
            } else {
5             j ← l;
6             while (j > 0 and x[j] = t[pos + j]) do j ← j − 1;
7             if (j = 0) then add pos to P;
8             pos ← pos + q;
            } end if
          } end while
          return (P);
        end function
```

FIG. 20. Constant-space algorithm in case of large period.

PROOF. One can get the first assertion by reproducing a simplified version of the proof of Proposition 2.2.

We first prove that the number of comparisons performed at line 3 is bounded by $|t|$. Consider two consecutive values k and k' of the sum pos $+ i$. If these values are obtained during the same execution of the instruction 3 then $k' = k + 1$ because i is increased by one unit. Otherwise, *pos* is increased either by the execution of instruction 4 or by the execution of instruction 8. In the first case, $k' = k - l + l + 1 = k + 1$ again. In the second case, $k' \geq k + q - l$, and the assumption $q > \max(l, |x| - l)$ implies $k' \geq k + 1$. Since comparisons at line 3 strictly increase the value of $pos + i$, which has initial value $|x_l| + 1$ and final value at most $|t| + 1$, the claim is proved.

We show that the number of comparisons performed at line 6 is also bounded by $|t|$. Consider two values k and k' of the sum $pos + j$, respectively, obtained during two consecutive executions of the instruction 6. Let p be the value *pos* has at the first of these two executions. Then, $k \leq p + l$ and $k' \geq p' = p + q$. The assumption $q > \max(l, |x| - l)$ implies $k' \geq k + 1$. Thus, no two letter comparisons at line 6 are done on a same letter of the text t, which proves the claim.

The total number of comparisons is thus bounded by $2|t|$. □

The complete two-way string-matching algorithm is shown in Figure 21. The function "MATCH" is obtained by substituting the bodies of functions "POSITIONS" and "POSITIONS-BIS" for statements at lines 5 and 6, respectively, inside the function "SMALL-PERIOD." At the beginning of the last "else" part, q is assigned to $\max(l, |x| - l)$. The next proposition sums up the results of the previous sections.

PROPOSITION 5.3. *On input words x and t, the function "MATCH" computes the set P(x, t) of positions of the pattern x inside the text t in time* $O(|t| + |x|)$. *More precisely, the computation uses less than* $2|t| + 5|x|$ *letter comparisons and* 13 *extra memory locations.*

PROOF. The correctness of the algorithm is a straightforward consequence of Proposition 2.2 and 5.2.

```
function MATCH (x, t):
  n ← LENGTH(x);
  (l1, p1) ← MAXIMAL-SUFFIX(x, ≤);
  (l2, p2) ← MAXIMAL-SUFFIX(x, ⊆);
  if (l1 ≥ l2) then {
    l ← l1; p ← p1;
  } else {
    l ← l2; p ← p2;
  } end if
  if (l < n/2 and x[1] ··· x[l] is a suffix of x[l + 1] ··· x[l + p]) then {
    P := ∅; pos ← 0; s ← 0;
    while (pos + n ≤ |t|) do {
      i ← max(l, s) + 1;
      while (i ≤ n and x[i] = t[pos + i]) do i ← i + 1;
      if (i ≤ n) then {
        pos ← pos + max(i − l, s − p + 1);
        s ← 0;
      } else {
              j ← l;
              while (j > s and x[j] = t[pos + j]) do j ← j − 1;
              if (j ≤ s) then add pos to P;
              pos ← pos + p;
              s ← n − p;
      } end if
    } end while
    return (P);
  } else {
    q := max(l, n − l) + 1;
    P := ∅; pos ← 0;
    while (pos + n ≤ |t|) do {
      i ← l + 1;
      while (i ≤ n and x[i] = t[pos + i]) do i ← i + 1;
      if (i ≤ n) then {
        pos ← pos + i − l;
      } else {
        j ← l;
        while (j > 0 and x[j] = t[pos + j]) do j ← j − 1;
        if (j = 0) then add pos to P;
        pos ← pos + q,
      } end if
    } end while
    return (P);
  }end if
end function.
```

FIG. 21. Two-way string-matching algorithm.

As shown at the beginning of this section, the overall precomputations of the period and of a critical position of x uses less than $5|x|$ comparisons. Propositions 2.3 and 5.2 assert that each search phase uses less than $2|t|$ comparisons, which gives the total number of comparisons $2|t| + 5|x|$.

The function "MAXIMAL-SUFFIX" requires seven memory locations including n and so do the search phases (see Corollary 2.4). We get 13 locations. No more locations are needed because $l1, l2$ can share the same place as l and so can $p1, p2$ and q with p. □

We end this section by giving examples of the behavior of the two-way algorithm of Figure 21. The examples are presented in Figure 22, where the

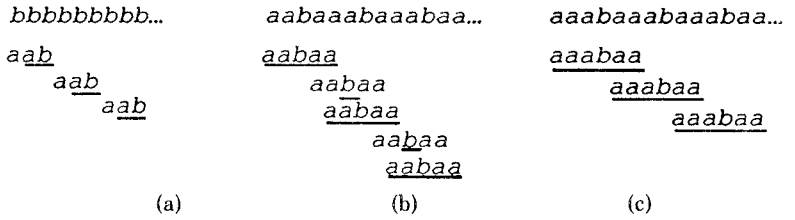

FIG. 22. Behavior of function "MATCH".

letters of the pattern scanned during the search phase of the algorithm are underlined.

On the pattern $x = a^n b$, the algorithm finds its unique critical factorization (a^n, b). The search for x inside the text $t = b^m$ uses $2 \mid t \mid / \mid x \mid$ comparisons and so does BM. Both algorithms attempt to match the last two letters of x against the letters of t, and shift $x(n + 1)$ places to the right, as shown in Figure 22(a).

The pattern $x = a^n b a^n$ has period $n + 1$. Its critical factorization computed by the function "MATCH" is (a^n, ba^n). The function behaves as "POSITIONS" and uses $2 \mid t \mid - 2e - 2$ comparisons to match x in $t = (a^n ba)^e a^{n-1}$ (see Figure 22(b)). The same number of comparisons is reached when searching for $x = a^n b a^{n-1}$ inside $t = (a^n b)^e a^{n-1}$, but in this latter case, the algorithm behaves as "POSITIONS-BIS" (see Figure 22(c)).

6. *Conclusion*

As a conclusion, the two-way algorithm exhibits a time complexity that is linear, as for KMP and BM, and that is, in some cases, sublinear as BM is. An average analysis remains to be done.

There is a version of BM that is frequently used [19, 28]. It uses an additional information to compute the shifts of the pattern, given a function

$$d(a) = \min(\{ \mid w \mid ; aw \text{ is a suffix of } x\} \cup \{ \mid x \mid \}),$$

defined on the letter a of A. Such a function can also be incorporated into our algorithm. According to a critical factorization (u, v) of x such that $\mid u \mid < p(x)$, we define a function α for the last occurrence of a letter inside the left part u of the critical factorization:

$$\alpha(a) = \min(\{ \mid w \mid ; aw \text{ is a suffix of } u\} \cup \{ \mid u \mid \}).$$

The statement at line 4 in the algorithm of Figure 8 is replaced by

$$pos \leftarrow pos + \max(i - l + \alpha[t[pos + i]], s - p + 1),$$

and the line 4 in Figure 20 by

$$pos \leftarrow pos + i - l + \alpha[t[pos + i]].$$

With similar transformations on the corresponding statements of the function "MATCH" of Figure 21, the minimal number of comparisons during the search phase becomes $\mid t \mid / \mid x \mid$, the best possible as shown in [25]. An instance of this best case is given by $x = a^n b$ and $t = c^m$. On the pair of words

$x = a^n ba^n$ and $t = (a^n c)^e$, the number of comparisons becomes less than $|t|/n$ compared to $|t| - n$ for BM.

KMP is one of the simplest examples of an algorithm that is in linear time without operating in real time. In fact, the time spent by the algorithm on a single symbol of the text cannot be bounded by a constant. The first string-matching algorithm operating in real time is due to Galil [16]. Another real time string-matching algorithm is presented in [11]. It is based on the notion of a suffix automaton invented by Blumer et al. ([6, 10]). The nature of the BM algorithm and the two-way algorithm does not allow them to operate in real time.

As already mentioned, the two-way algorithm uses constant extra memory space, even including the precomputation of the period of the pattern, as shown in Section 5. However, the algorithm uses two arrays to store the text and the pattern. KMP can be implemented without a memorization of the text since it behaves like a deterministic automaton receiving the text as input, symbol by symbol, without reading back its input. This can be considered as its main advantage compared to the naive algorithm. Our algorithm, as well as BM, does not present this feature and requires the use of two arrays of the same size, one for the pattern and another one used as a buffer on the text. It does not use, however, an auxiliary array for the shift function as KMP and BM do. It is similar in this sense to Galil and Seiferas' algorithm [17], and allows an implementation without dynamic allocation of memory.

From the point of view of automata theory, the problem of string-matching is a particular case of the computation of a rational relation. A rational relation between words is one that may be computed by a multihead finite-state automaton (see [5] for definitions and results on rational relations). It holds in general that one may verify in linear time whether two words are related by a given rational relation (see [5], Exercise 8.2). The possibility of realizing the string-matching in linear time and constant space has been interpreted in [17] as the possibility of implementing it with a multihead deterministic finite-state automaton. It would be interesting to know to which class of rational relations this results applies.

ACKNOWLEDGMENTS. The present version of this paper incorporates modifications to the first two versions suggested by the referees, who are gratefully acknowledged.

REFERENCES

1. AHO, A. V. Pattern matching in strings. In R. V. Book, ed., *Formal Language Theory: Perspectives and Open Problems*. Academic Press, Orlando, Fla., 1980, pp. 325–347.
2. AHO, A. V., AND CORASICK, M. J. Efficient string matching: An aid to bibliographic search. *Commun. ACM 18*, 6 (June 1975), 333–340.
3. APOSTOLICO, A., AND GIANCARLO, R. The Boyer–Moore–Galil searching strategies revisited. *SIAM J. Comput. 15*, 1 (1986), 98–105.
4. BEAN, D. R., EHRENFEUCHT, A., AND MCNULTY, G. F. Avoidable patterns in strings of symbols. *Pacific J. Math. 85* (1979), 261–294.
5. BERSTEL, J. *Transductions and Context-Free Languages*. Teubner, Stuttgart, Germany, 1979
6. BLUMER, A., BLUMER, J., EHRENFEUCHT, A., HAUSSLER, D., CHEN, M. T., AND SEIFERAS, J. The smallest automaton recognizing the subwords of a text. *Theoret. Comput. Sci. 40*, 1 (1985), 31–56.
7. BOOTH, K. S. Lexicographically least circular substrings. *Inf. Process. Lett. 10*, 4/5 (1980), 240–242.

8. BOYER, R. S., AND MOORE, J. S. A fast string searching algorithm, *Commun. ACM 20*, 10 (Oct. 1977), 762–772.
9. CESARI, Y., AND VINCENT, M. Une caractérisation des mots périodiques. *C.R. Acad. Sci.* 286, série A (1978), 1175.
10. CROCHEMORE, M. Transducers and repetitions *Theoret. Comput. Sci. 45* (1986), 63–86.
11. CROCHEMORE, M. Longest common factor of two words. In H. Ehrig, R. Kowalski, G. Levi, and U. Montanari, eds., *TAPSOFT'87*, vol 1. Springer-Verlag, 1987, pp. 26–36.
12. CROCHEMORE, M. String-matching and periods. *Bull. Euro. Assoc. Theor. Comput. Sci. 39* (1989), 149–153.
13. DUVAL, J. P. Mots de Lyndon et périodicité. *R.A.I.R.O. Informat. Theor. 14*, 2 (1980), 181–191.
14. DUVAL, J. P. Factorizing words over an ordered alphabet. *J. Algorithms 4* (1983), 363–381.
15. GALIL, Z. On improving the worst case running time of the Boyer–Moore string-matching algorithm. *Commun. ACM 22*, 9 (Sept. 1979), 505–508.
16. GALIL, Z. String matching in real time. *J. ACM 28*, 1 (Jan. 1981), 134–149.
17. GALIL, Z., AND SEIFERAS, J. Time space optimal string matching. *J. Comput. Syst. Sci. 26* (1983), 280–294.
18. GUIBAS, L. J., AND ODLYZKO, A. M. A new proof of the linearity of the Boyer–Moore string searching algorithm. *SIAM J. Comput. 9*, 4 (1980), 672–682.
19. HORSPOOL, N. Practical fast searching in strings. *Softw. Prac. Exp. 10* (1980), 501–506.
20. KARP, R. M., MILLER, R. E., AND ROSENBERG, A. L. Rapid identification of repeated patterns in strings, trees, and arrays. In *Proceedings of the 4th Annual ACM Symposium on Theory of Computing* (Denver, Colo., May 1–3). ACM, New York, 1972, pp. 125–136.
21. KNUTH, D. E., MORRIS, J. H., AND PRATT, V. R. Fast pattern matching in strings, *SIAM J. Comput. 6*, 2 (1977), 323–350.
22. LOTHAIRE, M. *Combinatorics on Words*. Addison-Wesley, Reading, Mass., 1982.
23. LANDAU, G. M., AND VISHKIN, U. Efficient string matching with k mismatches. *Theor. Comput. Sci. 43* (1986), 239–249.
24. MCCREIGHT, E. M. A space-economical suffix tree construction algorithm. *J. ACM 23*, 2 (Apr. 1976), 262–272.
25. RIVEST, R. L. On the worst-case behavior of string-searching algorithms. *SIAM J. Comput. 6*, 4 (1977), 669–674.
26. ROZENBERG, G., AND SALOMAA, A. *The Mathematical Theory of L Systems*, Academic Press, New York, 1980.
27. RYTTER, W. A correct preprocessing algorithm for Boyer–Moore string-searching. *SIAM J. Comput. 9*, 3 (1980), 509–512.
28. SEDGEWICK, R. *Algorithms*. Addison-Wesley, Reading, Mass., 2d ed., 1988.
29. SHILOACH, V. Fast canonization of circular strings. *J. Algorithms 2* (1981), 107–121.
30. SLISENKO, A. O. Detection of periodicities and string-matching in real-time. *J. Sov. Math. 22*, 3 (1983), 1316–1387.
31. THOMPSON, K. Regular expression search algorithm. *Commun. ACM 11*, 6 (June 1968), 419–422.
32. UKKONEN, E. Finding approximate patterns in strings, *J. Algorithms 6* (1985), 132–137.
33. VISHKIN, U. Optimal parallel pattern matching in strings. *Inf. Cont. 67* (1985), 91–113.
34. WEINER, P. Linear pattern matching algorithms. In *Proceedings of the 14th Annual IEEE Symposium on Switching and Automata Theory*. IEEE, New York, 1972, pp. 1–11.

RECEIVED FEBRUARY 1988; REVISED FEBRUARY AND AUGUST 1989 AND MARCH 1990; ACCEPTED APRIL 1990